博文金融

金融科技
丛书

The Science of the Blockchain

# 区块链
# 核心算法解析

[瑞士] Roger Wattenhofer 著
陈晋川 薛云志 林强 祝庆 译

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

## 译者简介

陈晋川，香港理工大学博士，中国人民大学信息学院副教授，硕士生导师，曾作为访问学者先后在微软亚洲研究院和德国乌尔姆大学工作。目前研究方向为大数据管理、区块链。

薛云志，中国科学院软件研究所博士，清华大学MBA，中国科学院软件研究所副研究员，硕士生导师，研究方向为人工智能、软件工程。

林强，律师、专利代理人，中国科学院软件研究所计算机应用硕士。执业领域为知识产权法，尤其是专利咨询、申请、管理和权利行使。于2004年加入北京东方亿思，一直致力于帮助许多财富500强跨国公司管理他们在中国的专利组合。近年来，还帮助一些互联网巨头和国内初创企业建立、管理全球专利组合。

祝庆，计算机科学硕士研究生，毕业于中国科学院研究生院。现任职于中国工商银行总行，之前在甲骨文Oracle、IBM、Teradata等公司担任首席企业架构师、项目总监等职位，在金融电信媒体行业有多年行业经验。


The Science of the Blockchain

# 区块链
# 核心算法解析

[瑞士] Roger Wattenhofer 著
陈晋川 薛云志 林强 祝庆 译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

版权贸易合同登记号　图字：01-2017-1717

**图书在版编目（CIP）数据**

区块链核心算法解析 /（瑞士）罗格·瓦唐霍费尔（Roger Wattenhofer）著；陈晋川等译. —北京：电子工业出版社，2017.8
（金融科技丛书）
书名原文：The Science of the Blockchain
ISBN 978-7-121-31328-8

Ⅰ. ①区…　Ⅱ. ①罗…　②陈…　Ⅲ. ①电子商务－支付方式－研究　Ⅳ. ①F713.361.3

中国版本图书馆 CIP 数据核字（2017）第 076211 号

责任编辑：高洪霞
印　　刷：三河市双峰印刷装订有限公司
装　　订：三河市双峰印刷装订有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：720×1000　1/16　印张：10.25　字数：151.2 千字
版　　次：2017 年 8 月第 1 版
印　　次：2017 年 8 月第 1 次印刷
定　　价：59.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819，faq@phei.com.cn。

# 推荐序 I

毫无疑问，互联网是 20 世纪最伟大的发明之一。随着信息、通信技术的蓬勃发展，互联网已渗透到生产、贸易、通信、学习、娱乐等人们生活的几乎所有方面，更使很多方面产生了革命性的变化。近十年来，在互联网的基础上，移动互联网、物联网，乃至智能互联网得到了新的发展。人工智能、深度学习、机器学习等一系列技术和理论的新发展，又促使互联网应用面临更加蓬勃发展的新局面。在众多的互联网新应用中，不得不提及区块链。

仿佛一夜之间，互联网创业圈和金融圈都在谈论区块链。坚信者认为，去中心化的、不可篡改的分布式账本，能够重构金融体系，甚至重塑整个社会。不知区块链之父当初是否曾预见到如今区块链的热度？

如今，比特币及其他虚拟货币已广泛流行，并且引起了监管当局的关注；政府、巨头和创业公司，也都积极参与到区块链的各种应用的探索中。然而，在互联网土壤上生长出的各种技术和应用中，区块链及其应用还很年轻。自 2009 年比特币诞生至今，也才仅 7 年，更不要说区块链在互联网金融领域和其他领域的应用。

作为一个一直关注新技术发展的互联网“老兵”，我曾数次应邀参加中关村区块链产业联盟的活动，和互联网领域的年轻创业者、专家、学者一起，探讨、推动区块链的发展和应用。我们的年轻人，尤其是年轻的创业者，他们的大胆探索和勇于创新，令我感到欢欣鼓舞。

目前，介绍区块链应用的书籍非常多，而从理论、技术层面介绍区块链的书比较少。很高兴看到有这样一本从理论、技术层面介绍区块链的书籍出版。希望大家能耐心读读这本书，更深入地理解区块链技术，从而有助于推动区块链技术的发展和应用。

高卢麟博士
中国互联网协会副理事长
美国芝加哥马歇尔法学院客座教授

# 推荐序 II

区块链（Block Chain）原本只是比特币网络的一种记账技术，近几年来却在金融、知识产权、数据交易、电子证照、慈善、新能源等领域引起了广泛的关注。为什么就突然“火”起来了？究其原因，我的理解是：第一，区块链具有去中心化的特征，不以参与交易的任何一方为中心。去中心化可以带来效率的提升和成本的降低，直接增加了企业的利润。第二，区块链具有去信任的特征，也就是假定参与交易的任何一方都不是可信任的。我们通过记录交易的信息，而且是不可抵赖的，来迫使交易各方遵守诚信。因此也可以说，区块链技术很好地回应了目前互联网的痛点——诚信问题。第三，区块链作为互联网的一种基础设施，也可以看作是一种分布式数据库，其核心就是参与交易的多方如何达成共识。在分布式数据库中，为了处理并发事务，需要在不同的节点上维护一个全局一致的状态，传统的做法是通过两阶段锁协议来实现。另一方面，通常大型应用会维护多个数据库副本，以实现数据库的恢复。在多个数据库副本之间维护一致的状态也是一个经典的难题，而解决这个难题的最佳算法实践正是本书中的重点内容——Paxos 算法。这个算法在大数据管理时代更是大放异彩，在 BigTable，Hadoop 等多个大数据计算平台上得到应用。

目前市场上关于区块链的书籍很多，但大多偏于介绍区块链的基础知识及应用前景，纯技术的书籍相对较少。本书着眼于区块链的核心问题——拜占庭共识，针对不同的应用场景，介绍了适用的分布式共识算法。书中包含了很多算法及证明，深入剖析了共识算法的核心思想。本书详细介绍了在不同应用场景下的分布式共识算法，包括单纯宕机错误（节点只可能发生宕机，但不会恶意犯错），拜占庭式错误节点（可

以认为是恶意的节点，呈现任何行为），允许消息签名，仲裁系统，弱一致条件下的共识等，并介绍了分布式存储的一些基础知识（如一致性哈希）。书中提到的很多算法，特别是 PBFT，目前是区块链的重要分支——联盟链的核心算法。

对于从事区块链的研究者或工程技术人员来说，共识算法是需要认真弄清楚的内容。虽然存在不少开源的共识算法或区块链框架，但不同的应用对共识算法的要求是不一样的，应该根据应用的特点选择合适的共识算法，甚至对已有的共识算法做必要的剪裁。要做到这一点，就必须理解基础的分布式共识算法。而这就是本书的最大价值。

本书译者之一，陈晋川博士，自 2009 年从香港理工大学毕业后加入中国人民大学，一直在我的研究团队里工作。在大数据、分布式数据管理等领域做出了不少优秀成果。晋川从去年开始关注区块链，他在查阅很多文献之后，发现关于区块链核心技术部分的资料很少，很难把握其精髓。在看到本书原著之后，他感觉这正是目前市场所缺少的干货，就决定将其翻译出来。本书内容艰深，为了准确、清晰地将内容译为中文，晋川投入了大量心血。作为一线科研工作者，能在背负巨大科研考核压力的情况下，投入如此多精力从事翻译工作，殊为不易。

其实，这本书不能算是严格意义上的翻译，译者除了原稿翻译之外，还增加了很多译者自己的注释，对书中的算法、公式进行注解（作者很多地方写得较为简略）。另外，书中还增加了两章新的内容。一章是介绍 Paxos 算法的发展史和在工业界的应用情况，另一章是对比分析当前主流的两个共识机制，比特币的 PoW 和私有链的 PBFT。现在都讲究“混搭”，这本译著也是一种形式的混搭。

杜小勇

中国计算机学会数据库专委会主任

教育部数据工程与知识工程重点实验室主任

# 推荐序 III
# 智能时代的区块链

“身为一个智能时代的潮人，谁的口袋里还不装着几块比特币？”

近年来，比特币作为第一种数字加密货币，受到了褒贬不一的评价；其价格一路飞涨，但走向主流货币之路却是“路漫漫其修远兮”。但是，作为比特币的核心底层技术，区块链技术的受关注程度却一路飙升。自2014 年以来，国内外多家著名机构在这一研究方向上已经累计投入了数百亿美元。其研发范围已经远远超越单纯的数字加密货币，遍及银行、保险、物流、物联网、资产交易、公证与鉴别、社交通信等领域。甚至有人宣称，未来的社会及政府都可以架构于区块链之上。

区块链为何会受到如此广泛的关注？究其原因，其核心特征，去中心化、去信任化、智能合约等，正好满足未来互联网持续发展所要求的信息的高度自动化和高度程序化的安全流转需求。我们知道，互联网作为一个连接媒介，连接了世界上一部分的信息、数据、交互和交易；移动互联网更是将连接范围进一步扩大至几乎所有人类活动的范围，基本完成了人类活动的数字化。

在技术驱动下，人类生活方方面面的数据量开始激增，大规模计算出现了分布化和并行化的特点。在此基础上，人工智能第三次浪潮汹涌而来，进一步衍生了对高度自动化和高度程序化的信息流转的需求：非自动化环节必然被不断削减；信息的审核和控制将不再受限于任何一个权威性的中心节点，也不应该被任何一个参与节点伪造、篡改和否认。对于上述需求，区块链技术可以相对完美地提供解决方案。因此，可以

预见，区块链技术将会是智能时代数据存储、传输、分发最有竞争力的解决方案之一。

区块链作为一个分布式的数据存储、传输和分发解决方案，其核心在于如何在一个分布式、无受信节点的环境下，自动形成共识（consensus）的机制。所谓共识机制，是指分布式系统中全部节点（或者大部分节点）就某条数据的真实性或者某条交易的价值达成一致，并据此更新各节点记录的一种机制。不难看出，面对不同的场景、不同的应用需求，需要设计并实施不同的共识机制。这就需要对共识机制在理论层面、技术层面有深入的理解。在智能时代，区块链参与节点将可能是一个智能体，区块链系统也将可能是一个多智能体系统。系统中节点的数量和行为可能是基于人工智能技术自动产生和调整，而不是基于程序员所编写的固定规则。这种情况下，要想利用共识机制充分保证区块链的安全运行，需要对共识机制更为深刻的理解。

国内关于区块链的书籍已经有很多了，但大多都是在谈应用、谈理念或者在谈相关的投融资，真正涉及技术细节的书籍相对较少。《区块链核心算法解析》以共识机制为主体，系统介绍了区块链所涉及的各种关键定理和证明，也给出了相应算法。难能可贵的是，作者还结合实例讲述了不同场景下的共识机制的设计方法。这是一本关于区块链核心技术的系统论著，对于区块链科研和应用人员都具有很高的参考价值。

戴斌

国防科技大学机电工程与自动化学院副总工程师

# 前　言

当你和从事金融科技（FinTech）的朋友在一起交谈时，不可避免地会注意到一个非常流行的词——区块链。听上去，它就像 Internet 那样重要。某些金融科技领域的朋友认为区块链是一段神奇的代码，它可以使一个分布式系统的参与者们就系统的状态达成共识，从而追踪系统的变化。区块链这个词语是从比特币借用的，然而，早在区块链或者比特币出现之前，共识技术（或协定技术）（Agreement Techniques）就已经在分布式系统领域中存在了。曾经出现过各种各样的概念和协议，各有其优点和缺点。

本书的目的是深入剖析这项当前最引人注目的技术——区块链。如果你是一名开发者（不管是否在金融科技领域），本书将帮助你更好地理解：在你研发的分布式系统中，什么是对的，什么是错的，什么是可能的，而什么是不可能的。

## 内容简介

本书介绍了构建容错的分布式系统所需的基础技术，以及一系列允许容错的协议和算法，并且讨论一些实现了这些技术的实际系统。

本书中的主要概念将独立成章。每一章都以一个小故事开始，从而引出该章节的内容。算法、协议和定义都将以形式化的方式描述，以便于读者理解如何实现。部分结论会在定理中予以证明，这样读者就可以明白为什么这些概念或算法是正确的，并且理解它们可以确保实现什么。其他的大部分内容将以评论的方式出现。这些评论将讨论各种各样

非正式的思考，并且为后续内容做好铺垫。就算不阅读这些评论，读者们也可以掌握章节的精髓。此外，为了便于读者寻根溯源，每一章也会讨论相关技术的发展历史。

本书将介绍不同的模型 (以及模型的组合)，以适用于不同的场景。本书关注的是实用的协议和系统。换句话说，我们在选择概念时，不会根据这些概念是否看起来有意思，而是根据它们是否有实际的价值。

不管怎样，希望你在本书中找到乐趣！

# 目　录

# 第 1 章　绪论

## 1.1　分布式系统是什么

今天的计算机系统和信息系统在本质上都是分布式的。越来越多的公司进入全球化时代，它们拥有部署在不同大陆上的成千上万的计算机。数据存储在不同的数据中心，而计算任务则运行在多台计算机上。

另一方面，我们的智能手机也是一个分布式系统。这不仅因为它很可能将你的数据存储在云端，也因为它本身就包含多个处理和存储单元。

此外，计算机的发展已经经历了一个很长的过程。在 20 世纪 70 年代早期，微型集成电路芯片的时钟频率只有 1MHz 左右。10 年之后，在 20 世纪 80 年代早期，个人电脑的主频已经达到了 10MHz。到了 1990 年，时钟频率达到了 100MHz。到了 2000 年，消费者们已经用上了 1GHz 的处理器。仅仅几年之后的 2005 年，家用电脑的主频已经在 3GHz 到 4GHz 之间。然而，今天我们在市场上见到的个人电脑时钟频率依旧徘徊在 3GHz 到 4GHz，因为 CPU 时钟频率已经在 2004 年前后基本停止增长。如果不能解决一些物理上的问题，比如过热，时钟频率将无法再显著提升。

简而言之，今天几乎所有的计算系统都是分布式的，原因如下。

- 地理因素：大的公司和组织必然分布在多个地方。
- 并行化：我们需要使用多核处理器或计算机集群来加速计算。

- 可靠性：数据需要备份在不同的机器上以避免丢失。
- 可用性：数据需要复制到不同的机器上以利于快速获取，避免可能的瓶颈，并减少延迟。

虽然分布式系统带来了很多好处，比如扩大存储容量和计算能力，甚至有可能连接地理空间上分离的区域，然而它也带来了一个很麻烦的协调问题 (Coordination Problems)。协调问题非常普遍，具备不同的特点，也有着不同的称谓，诸如：区块链 (Blockchain)、一致性 (Consistency)、协定 (Agreement)、共识 (Consensus)、账单 (L edger)、事件溯源 (Event Soucing) 等。

在分布式系统中，协调问题是很常见的。即便是一个分布式系统中的每个节点 (如计算机、核、网络交换机等) 几年才发生一次故障，但如果系统包含数百万个节点，那么平均每分钟都将发生一次故障。从好的方面讲，人们期待一个多节点的分布式系统可以容忍一些错误并且持续正常工作。

## 1.2 本书概览

本书的核心概念将在第 2 章介绍，即定义 2.8 中的状态复制 (State Replication)。如果一个分布式系统的所有节点在一个命令序列上取得共识 (即同样一组命令按同样的顺序执行)，那么我们就可以得到状态复制。在金融技术领域，状态复制常常等同于区块链。状态复制可以通过不同的算法来获得，具体取决于系统能够容忍的错误类型。

在第 2 章中，我们将介绍基本的定义，并且介绍 Paxos 算法。即使系统中少部分节点崩溃 (Crash)，Paxos 算法也可以获得状态复制。

在第 3 章中，我们将明白 Paxos 算法可能无法成功。事实上如果

我们不走运，将没有一个确定性的协议 (Deterministic Protocol) 能解决状态复制。然而，我们也会介绍一个快速的随机共识协议。即使有崩溃性错误，该协议也能得到状态复制。

在第 4 章中，将目光扩展到简单的崩溃性错误之外。我们将介绍一些应对恶意行为 (Malicious Behavior) 的协议，它们运行在同步或异步的系统中。此外，我们将介绍对正确行为 (Correct Behavior) 的不同定义。

在第 5 章中，我们会使用一个密码学概念，消息认证 (Message Authentication)。首先介绍一个简单的同步协议，接着介绍 Zyzzyva 协议，这是当前最好的异步协议。当消息认证可用时，这个协议能够实现状态复制。

在第 6 章中，我们通过研究所谓的仲裁系统 (Quorum Systems) 来分析可扩展性问题。当集群的算力不足，且不能靠增加新节点来解决问题时，仲裁系统或许是一个优雅的解决方案。

在第 7 章中，我们介绍弱一致 (Weaker Consistency) 的概念，并且以比特币协议为例进行详细分析。

最后，我们在第 8 章中介绍了一些更弱的一致性概念，并且介绍了高可扩展性分布式存储系统 (Highly Scalable Distributed Storage) 的解决方案。

## 章节说明

关于分布式系统共识问题，已有不少很好的教科书，例如 [AW04, CGR11, CDKB11, Lyn96, Mul93, Ray13, TS01]。James Aspnes 还编写了一个非常好的关于分布式系统的免费教材 [Asp14]。与本书类似，这

些教材也主要着眼于大规模分布式系统，因此和这本书在内容上有所重叠。此外，还有不少优秀的教材着眼于小型的多核系统，比如 [HS08]。

一些同事在本书写作过程中提供了大量帮助，在此一并予以致谢：Pascal Bissig, Philipp Brandes, Christian Decker, Klaus-Tycho Förster, Barbara Keller, Rik Melis, 以及 David Stolz (以上排名按字典顺序)。

# 参考文献

[Asp14] James Aspnes. Notes on Theory of Distributed Systems, 2014.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition)*. John Wiley Interscience, March 2004.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mul93] Sape Mullender, editor. *Distributed Systems (2Nd Ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer Publishing Company, Incorporated, 2013.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

# 第 2 章　容错问题和 Paxos 算法

我们应该如何创建一个容错的分布式系统呢？本章将从一些简单的问题开始，一步步改进我们的方案，最终将得到 Paxos，一个甚至可以在逆境下工作的协议。

## 2.1　客户端/服务器

**定义** 2.1 (**节点** (Node)). [1] 系统中一个工作单元被称为节点 (Node)。在一个计算机网络中，所有的计算机都是节点。在经典的客户端/服务器模式下，服务器和客户端都是节点。本文中，如果不另外说明，系统中节点的总数为 $n$.

**模型** 2.2 (**消息传递** (Message Passing)). 我们在消息传递模式 (Message Passing Model) 下研究由一组节点构成的分布式系统。每个节点都能进行本地运算，并可以向所有其他节点发送消息。

**评论:**

- 我们从最小的分布式系统 (仅包含两个节点) 开始。该系统中有一个客户端节点，它希望操作 (比如存储或更新) 在远程服务器节点上的数据。

[1] 译者注：原著中将定义、模型、算法等统一编号，为保持风格一致，译文中采用了相同的编号方式。

**算法 2.3** 朴素的客户端/服务器算法

1: 客户端每次向服务器发送一条命令

**模型 2.4** (**消息丢失** (Message Loss)). 在存在消息丢失 (Message Loss) 的消息传递模式下，任何一条消息都不能保证可以安全地到达消息的接收者。

**评论:**

- 一个相关的问题是消息损坏，即收到了一条消息，但是其内容已经损坏了。实际上，与消息丢失相比，我们可以更好地处理消息损坏，比如在消息中增加校验码。
- 如果消息丢失，则算法 2.3 不能正常工作。因此我们需要一点小改进。

**算法 2.5** 带确认 (Acknowledgements) 的客户端/服务器算法

1: 客户端每次向服务器发送一条命令
2: 服务器每收到一条命令，都会发送一条确认信息
3: 如果客户端没有在一个合理的时间内收到确认信息，它将重新发送命令

**评论:**

- “每次发送一条命令”意味着如果一个客户端发送了一条命令 $c$，那么在它收到服务器对 $c$ 的确认信息之前，它将不会发送任何新的命令 $c'$。
- 不但客户端发送的消息可能丢失，服务器发送的确认消息也可能丢失。如果一条确认信息丢失，那么客户端可能重新发送一

条消息，即使该消息已经被服务器接收且执行。为了避免重复执行相同的消息，我们可以给每条消息加上序列号，这样接收者可以辨识出重复的消息。

- 这个看似简单的算法是很多可靠协议的基础，比如 TCP。
- 算法可以很容易地扩展到多个服务器的场景：客户端发送一条命令给每个服务器，一旦客户端收到所有服务器的确认消息，就可以认为该命令已被成功执行。
- 但是如何处理多个客户端的情况呢？

**模型** 2.6 (**可变消息延迟** (Variable Message Delay)). 在实际应用中，消息传输花费的时间是不同的。即使是在相同的一对节点间传输消息，所耗费的时间也可能不同。

**评论:**

- 本章假定都是可变消息延迟模式。

**定理** 2.7. 如果算法 2.5 在多个服务器和多个客户端间运行，服务器接收到的命令顺序可能是不同的，这将导致不一致的状态。

证明. 假定我们有两个客户端 $u_1$ 和 $u_2$，以及两个服务器 $s_1$ 和 $s_2$。两个服务器上都在维护同一个变量 $x$ 的值，起始状态，$x = 0$。两个客户端都在向服务器发送命令去更新 $x$ 的值。$u_1$ 的命令是 $x = x + 1$，而 $u_2$ 的命令是 $x = 2 \cdot x$。假设两个客户端同时发送它们的命令。因为有消息延迟，有可能 $s_1$ 先接收到 $u_1$ 的命令，而 $s_2$ 先收到 $u_2$ 的命令[2]。于是，在 $s_1$ 上执行两个命令的结果是 $x = (0 + 1) \cdot 2 = 2$，而在 $s_2$ 上计算的结果则是 $x = (0 \cdot 2) + 1 = 1$。

[2]比如，$u_1$ 和 $s_1$ 在地理位置上距离很近，且 $u_2$ 和 $s_2$ 也是如此。

**定义** 2.8 (**状态复制** (State Replication)). 对于一组节点，如果所有节点均以相同的顺序执行一个 (可能是无限的) 命令序列 $c_1, c_2, c_3, \ldots$，则这组节点实现了状态复制 (State Replication)。

**评论:**

- 状态复制是分布式系统的基本性质。
- 对于金融科技行业的从业人员来说，状态复制经常等同于区块链。在第 7 章中，我们将讨论比特币区块链，这实际上是实现状态复制的一种方式。我们也将在其他章节看到，还有很多值得认识的类似概念，它们有着不同的特点。
- 因为单个服务器可以天然实现状态复制，我们可以将单个服务器视为一个 串行化器 (Serializer)。通过让串行化器来分发命令，自动对请求进行排序并获得状态复制。

**算法 2.9** 借助单一串行化器实现状态复制

1: 所有的客户端都向串行化器发送命令，每次发送一条
2: 串行化器将命令逐条转发给所有的服务器
3: (针对某条命令) 一旦串行化器收到所有的确认信息，它通知 (发送该命令的) 客户端该命令已被成功执行

**评论:**

- 这个想法有时也被称为主—从复制 (Master-Slave Replication)。
- 但是如何处理节点故障呢？很显然，串行化器是一个潜在的单点故障 (Single Point of Failure)。

- 我们是否可以构造一个更分布式的方法来解决状态复制？与其直接构造一个一致的命令序列，不如换一个思路：想办法确保在任何时候最多只有一个客户端在发送命令。也就是说，我们采用互斥 (Mutual Exclusion) 和各自加锁 (Respectively Locking) 的思想。

**算法 2.10** 两阶段协议 (Two-Phase Protocol)

阶段 *1*

1: 客户端向所有的服务器请求锁

阶段 *2*

2: **if** 如果该客户端成功获得了所有服务器的锁 **then**
3: 该客户端以可靠的方式向每个服务器发送命令，随即释放锁
4: **else**
5: 该客户端释放已经获得的锁
6: 该客户端等待一段时间，再重新进入阶段 1
7: **end if**

**评论:**

- 这个想法曾出现在多个领域中，也有着不同的名称，比如两段锁协议 (Two-Phase-Locking) (2PL)。
- 另一个例子是两阶段提交协议 (Two-Phase Commit) (2PC)，典型场景是数据库系统。第一阶段被称为事务的准备阶段，第二阶段中这个事务或者提交 (Committed) 或者撤销 (回滚) (Aborted)。两阶段提交过程并非由客户端启动，而是在一个被选定的服务器上完成，这个服务器节点通常被称为协调者。

- 一般认为，如果节点可以在宕机之后恢复，较之一个简单的串行化器，2PL 和 2PC 能提供更好的一致性保证。特别对在节点宕机之前就启动的事务来说，存活的节点或许能和宕机的节点保持一致。在使用了一个额外阶段 (3PC) 的改进版协议中，这个优点更为明显。
- 2PC 和 3PC 的问题是，它们没有很好地处理异常。
- 算法 2.10 真的能很好地应对节点崩溃吗？不是！实际上，它甚至比那个简单的序列器方法 (算法 2.9) 更糟。算法 2.9 只要求一个节点必须正常工作，但是算法 2.10 要求所有服务器能够正常响应请求。
- 如果我们仅仅得到一部分服务器的锁，算法 2.10 能否工作？获得过半数节点的锁是否就足够了？[3]
- 如果两个或更多的客户端同时企图获得大部分服务器的锁，会发生什么情况？客户端是否必须放弃它们已经获得的锁，以避免死锁？怎么做？如果客户端在释放锁之前就发生故障，又该怎么办？我们是否需要一个与锁稍微不同的概念？

## 2.2 Paxos

**定义 2.11 (票 (Ticket))**. 一张票 (Ticket) 是一个弱化形式的锁，具备下面的性质。

- **可重新发布：** 一个服务器可以随时发布新的票，哪怕前面发布的票还没有被释放。

[3]译者注：是的，若一个客户端节点得到过半数服务器的锁，其他节点就不能再获得大数据节点的锁了。

- **票可以过期:** 当客户端使用一张票 $t$ 来给服务器发送消息时，仅当 $t$ 是最新发布的票时，服务器才会接收。

**评论:**

- 宕机问题被顺利解决：如果一个客户端在得到一个票之后宕机了，其他的客户端不会受到影响，因为服务器会发布新的票。
- 票可以使用计数器来实现：每当服务器收到一个 (发布票的) 请求时，将计数器加 1。这样当客户端尝试使用某个票时，服务器可以判定该票是否已经过期。
- 我们如何使用票？我们能简单地将算法 2.10 中的锁用票代替吗？我们需要增加至少一个额外阶段，因为只有客户端知道在阶段 2 中是否有过半数票是有效的。

**评论:**

- 该算法[4]是有问题的。假设 $u_1$ 是第一个成功地在过半数服务器上存储了命令 ($c_1$) 的客户端。但是 $u_1$ 很慢，在它告知所有服务器执行命令时 (第 9 行 [5])，另一个客户端 $u_2$ 在部分服务器上将命令更新为 $c_2$。然后，$u_1$ 告诉所有的服务器执行所存储的命令。此时，部分服务器将执行 $c_1$，而另一部分将执行 $c_2$。

[4]译者注: 与锁不同，票只是检查服务器当前是否空闲。得到票并不意味着服务器会为客户端保留位置，客户端必须马上去竞争。另外在理解算法时，须清楚本章假定的是可变消息延迟模式 (模式 2.6)。与算法 2.3 相比，本算法引入了竞争，多个客户端需要竞争得到过半数服务器的认可 (算法第 8 行)，这样降低了服务器执行不同命令序列的可能。此外，与算法 2.9 相比，本算法不需要串行化器，避免了单点故障，也可以在少量服务器宕机的情况下运行。但是，本算法仍然是**朴素**的，还存在明显的问题。后续将通过分析本算法的问题和相应的解决方案来引出 Paxos。

[5]译者注：原书此处写的是第 7 行。错了，应是第 9 行。

**算法 2.12** 朴素的基于票的协议

阶段 *1*

1: 客户端向所有的服务器请求一张票

阶段 *2*

2: **if** 收到过半数服务器的回复 **then**
3: 客户端将获得的票和命令一起发送给每个服务器
4: 服务器检查票的状态，如果票仍然有效，则存储命令并给该客户端一个正反馈消息
5: **else**
6: 客户端等待，并重新进入阶段 1
7: **end if**

阶段 *3*

8: **if** 客户端从过半数服务器处得到了正反馈 **then**
9: 客户端告诉所有的服务器执行之前存储的命令
10: **else**
11: 客户端等待，然后重新进入阶段 1
12: **end if**

- 如何解决这个问题呢？我们知道如果要修改 $u_1$ 存储在服务器上的命令，$u_2$ 必须使用比 $u_1$ 更新的票。因此，当 $u_1$ 的票在阶段 2 被接受之后，$u_2$ 必须在 $u_1$ 在服务器上存储命令 (第 4 行) 之后再拿到它的票。[6]
- 一个想法：如果在阶段 1 中，一个服务器不但发布票，而且也

[6]译者注：如果 $u_2$ 在 $u_1$ 存储命令之前就得到了它的票，则 $u_1$ 的存储行动将会失败，因为服务器会发现 $u_1$ 的票失效了。这样在阶段 3，$u_1$ 就不会要求服务器执行命令。

发布它当前所存储的命令。那么 $u_2$ 就知道 $u_1$ 已经存储了命令 $c_1$。$u_2$ 可以不要求服务器存储命令 $c_2$，而是继续存储 $c_1$。这样，两个客户端都尝试存储和执行相同的命令，那么谁先谁后就不再是一个问题。

- 但是，服务器们所存储的命令不一定相同，那么在阶段 1，$u_2$ 就可能从不同的服务器获知了多个不同的命令。它到底应该支持哪一个呢？
- 注意到支持最新存储的命令总是安全的。只要还不存在一个过半数服务器一致支持的命令，客户端们就可以支持任何命令。然而，一旦有一个过半数服务器一致的命令，所有客户端就必须支持这个命令。
- 因此，为了判定哪一个命令是最新存储的，服务器们必须记录存储命令所使用的票的编号，并且在阶段 1 把命令和相应的编号都告诉客户端。
- 如果每个服务器使用自己的票号，最新的票号就不一定是最大的。如果客户们自己来产生票号，那么这个问题可以解决！[7]

[7]译者注：我们需要一个全局一致的票号，因此不能让每个服务器自己维护一个本地的计数器来产生票号。一个巧妙的办法是让客户端自己来决定下一个票的编号 $t$，然后去向所有的服务器请求编号为 $t$ 的票。服务器在收到请求后，先将 $t$ 和它本地的计数器进行比较，只有 $t$ 大于本地计数器的值时，服务器才会发布票 (编号为 $t$)，同时将其本地计数器的值更新为 $t$。这样，我们就可以得到一个符合定义 2.11 的要求，且全局一致的产生票号的方法。

**算法 2.13** Paxos

| 客户端 (提案者) | 服务器 (接收者) |
| --- | --- |
| *初始化* | |
| $c$ ◁ 等待执行的命令 | $T_{\max} = 0$ ◁ 当前已发布的最大票号 |
| $t = 0$ ◁ 当前尝试的票号 | |
| | $C = \perp$ ◁ 当前存储的命令 |
| | $T_{\text{store}} = 0$ ◁ 用来存储命令 $C$ 的票 |
| *阶段 1* | |
| 1: $t = t + 1$ | |
| 2: 向所有服务器发消息，请求得到编号为 $t$ 的票 | |
| | 3: **if** $t > T_{\max}$ **then** |
| | 4: $T_{\max} = t$ |
| | 5: 回复: ok($T_{\text{store}}$, $C$) |
| | 6: **end if** |
| *阶段 2* | |
| 7: **if** 过半数服务器回复 ok **then** | |
| 8: 选择 $T_{\text{store}}$ 值最大的 ($T_{\text{store}}$, $C$) | |
| 9: **if** $T_{\text{store}} > 0$ **then** | |
| 10: $c = C$ | |
| 11: **end if** | |
| 12: 向这些回复了 ok 的服务器发送消息: propose($t$, $c$) | |
| 13: **end if** | |
| | 14: **if** $t = T_{\max}$ **then** |
| | 15: $C = c$ |
| | 16: $T_{\text{store}} = t$ |
| | 17: 回复：success |
| | 18: **end if** |
| *阶段 3* | |
| 19: **if** 过半数服务器回复 success **then** | |
| 20: 向每个服务器发送消息: execute($c$) | |
| 21: **end if** | |

**评论:**

- 与前面的算法不同，这个算法中没有明确地标出在哪个位置客户端可以跳转到阶段 1 并且开始新的尝试。实际上这并不是必要的，因为一个客户端可以在算法的任何位置取消当前的尝试并且开始新一轮尝试。这样的方式 (不明确标出何时开始新的尝试) 让我们不需要操心如何选择合适的超时 (timeout) 值。我们现在更关心正确性，而正确性和什么时候开始新的尝试是独立的。
- 在阶段 1 和阶段 2，如果票已过期，可以让服务器发送负的反馈，这样可以提高性能。[8]
- 连续两次尝试之间的等待时间可以用随机函数确定，这样可缓和不同节点之间的竞争。

**引理** 2.14. 我们将客户端发送的一条消息 propose($t$, $c$) (第 12 行) 称为一个内容是 $(t, c)$ 的提案。如果一项提案 $(t, c)$ 被存储在过半数服务器上 (第 15 行)，则称该提案被选中。如果已经存在一个被选中的 propose($t$, $c$) ，则对于后续每一个 propose($t'$, $c'$)，$c' = c$ 将始终成立 $(t' > t)$。

证明. 对于每一个票号 $\tau$, 最多只能有一个提案。根据算法 (第 2 行)，客户端先向所有服务器发送消息，请求编号为 $\tau$ 的票。而只有在收到过半数服务器对编号为 $\tau$ 的这张票的 ok 回复之后，客户端才会发送一个提案 (第 7 行)。因此每个提案都可以用它对应的票号 $\tau$ 来唯一标识。

[8]译者注：按上面的算法，在第 7 行，客户端必须等待过半数服务器的正反馈，直到超时才启动新一轮尝试。如果服务器发送负反馈，客户端可以更容易判定是否能够收到过半数的正反馈。

下面我们用反证法来证明。假设至少存在一个提案 propose($t'$, $c'$)，满足 $t' > t$ 且 $c' \neq c$。对于这样的提案，我们不妨只考虑那个拥有最小票号的提案，并假设该编号为 $t'$。因为 propose($t$, $c$) 和 propose($t'$, $c'$) 都已经被送达了过半数服务器，那么这两个过半数服务器集合之间必然存在一个非空交集 $S$，在 $S$ 中的服务器都收到了这两个提案。由于 propose($t$, $c$) 已经被选中，则至少有一个在 $S$ 中的服务器 $s$ 已经存储了命令 $c$。注意到当命令 $c$ 被存储时，票号 $t$ 仍然是有效的。因此，$s$ 必然是在存储 propose($t$, $c$) 之后才收到了关于票号 $t'$ 的请求，而且该请求使得票号 $t$ 失效。

于是，发出 propose($t'$, $c'$) 的客户端必然已经从 $s$ 处得知：某个客户端之前已经存储了 propose($t$, $c$)。根据算法第 8 行，每个客户端必须采纳已经存储且具有最高票号的命令，于是该客户端将提议 $c$ 而不是 $c'$。根据算法，只有一种可能使得该客户端不采纳 $c$：如果该客户端从某个服务器得知另一个提案 propose($t^*$, $c^*$) 已经被存储，且 $c^* \neq c$、$t^* > t$。但是，在这种情况下，一定存在一个客户端已经发送了提案 propose($t^*$, $c^*$)，且 $t < t* < t'$。于是这就与我们的假设矛盾："$t'$ 是所有在 $t$ 之后发布的提案中最小的票号"。

**定理** 2.15. 如果一条命令 $c$ 被某些服务器执行，那么所有的服务器最终都将执行命令 $c$。

证明. 根据引理 2.14 我们得知一旦一个关于包含 $c$ 的提案被选中，后续的每个提案都将采纳 $c$。由此可见，所有成功的提案都将采纳相同的命令 $c$。这样，只有采纳了命令 $c$ 的提案会被选中。此外，由于客户端只会在一条命令被选中之后告诉服务器执行该命令 (第 20 行)，每个客户端将最终告知所有的服务器执行命令 $c$。

**评论:**

- 如果拥有第一个成功提案的客户端没有宕机，它将直接告诉所有的服务器执行命令 $c$。
- 但是，如果客户端在告诉任何一个服务器执行命令之前就宕机了，服务器们将只有等到下一个客户端成功获得提案之后才可以执行命令[9]。一旦一个服务器接收到一个请求去执行命令，它可以通知所有后面到达的客户端：已经有一条命令被选中了。这样客户端就可以避免 (再向这个服务器) 继续发送提案。
- 如果超过一半的服务器宕机，Paxos 将不能工作。因为客户端不能再取得过半数服务器认可。
- 最初版本的 Paxos 包含三个角色：提案者、接受者，以及学习者。学习者不做任何事情，只是从其他节点学习哪个命令被选中了。
- 我们只让每个节点承担一个角色。在某些场景下，一个节点可能承担多个角色。比如，在一个 P2P 的场景下，每个节点既是服务器又是客户端。
- 上述算法必须信任客户端们 (提案者们) 会严格遵守协议。然而，这个假设在很多场景下并不合理。在某些场景下，提案者的角色可以被一组服务器承担，客户端们需要联络提案者，并用它们的名义来发布提案。
- 到现在为止，我们仅仅讨论了如何通过 Paxos 来使一组节点达成一致性决议 (decision) 来执行一条命令。单独的一条决议被称为 Paxos 的一个实例 (instance)。

[9]译者注：依然是前面宕机那个客户端所存储的命令。

- 如果希望执行多条命令，我们可以给每个实例附加一个实例编号。在每条消息中，该实例编号都会被使用。一旦某条命令被选中，任何一个客户端都可以采用一个新的实例编号来启动一个新的实例。如果一个服务器不知道前面的一个实例已经有了一个一致决议，那么该服务器可以询问其他服务器决议的内容。

## 章节说明

两阶段协议已经存在了很长时间，并且不止一个原创者。在 Gray 的书 [Gra78] 中，可以找到关于这个概念的较早的描述。

Leslie Lamport [10]在 1989 年提出了 Paxos。但是，“Paxos” 这个名称是怎么来的呢？Lamport 在论文中描述了一个虚拟的希腊小岛——Paxos，并假定这个算法是为了解决在岛上的议会中的投票问题。他非常喜欢这个想法，甚至以一个印第安纳·琼斯风格的考古学家的身份做过几个报告。当论文第一次投稿的时候，大多数评委都对这个议会故事感到困惑，无法理解算法的目的和思想。论文因此被拒绝发表！Lamport 拒绝修改论文，稍后他表示“不开心，因为这个领域的学者如此缺乏幽默感”。几年之后，随着应用的发展，Paxos 协议的用途变得更为广泛。Lamport 于是从抽屉里翻出这篇被拒绝的论文，并发给他的几位同事。这些同事们都表示很喜欢这个想法。于是 Lamport 决定重新投稿。和 8 年前的初稿相比，论文基本没改！但是这次论文被录用了。

不过，鉴于这篇论文 [Lam98] 确实太难懂了，Lamport 后来特地写了一篇关于 Paxos 的说明 [Lam01]。

[10]译者注：2013 年度图灵奖得主，LaTeX 的发明者，分布式计算领域最伟大的科学家之一。

# 参考文献

[Gra78] James N Gray. *Notes on data base operating systems*. Springer, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

# 延申阅读：Paxos 漫谈

Leslie Lamport (http://www.lamport.org/) 所提出的 Paxos 算法是现代分布式系统中的一项重要的基础性技术，得到了广泛的应用。**本章为译者编写，旨在帮助读者了解 Paxos 算法的发展过程，不同的版本，以及目前的应用情况。**

## 发展史

Paxos 的整个发展过程大概可以分为三个阶段。

第一阶段，萌芽期，大致是 1988—1996 年。1988 年，Liskov 等人在 PODC (ACM Symposium on Principles of Distributed Computing) 上发表了一篇文章 [OL88]，提出了一个在副本出现宕机情况下仍能正常工作的主从备份算法，该算法与 Paxos 在本质上是一致的 [Lam01]。

Leslie Lamport 在 1989 年提出了 Paxos 这个名称，但他的论文因为过于艰涩，未能发表。Lamport 企图用一个寓言故事来描述他的算法思想，却没有得到评委的认同。这篇文章直到 1998 年才正式发表 [Lam98]，那时分布式共识问题已经引起了广泛重视。从时间上看，Liskov 的论文更早一些，但是由于种种原因，特别是图灵奖得主 Butler Lampson 的推崇，Lamport 的工作影响力更广。

后来学界和工业界也基本都采纳了 Paxos 这个名称。值得一提的是，Liskov 在分布式共识领域同样做出了巨大的贡献。比如，目前风靡一时的 PBFT 算法 [MC99] 就出自 Liskov 的研究团队。Lampson 将这

两篇都归为“基础 Paxos”[Lam01]，以与后续其他版本的 Paxos 区分开来。在这个时期，虽然陆续有不少工作跟进，但整体仍处于不温不火的状态。

第二个阶段，即 1996—2007 年，这个时期可以用花团锦簇来形容。Paxos 引起了很多学者关注，涌现出一批 Paxos 的不同版本，这些 Paxos 的变种从不同侧面完善了基础 Paxos 算法，提升其性能，以符合不同应用的需要。1996 年，Butler Lampson 发表一篇论文“How to Build a Highly Availability System using Consensus”[Lam96]。在这篇论文里，Lampson 引用了大量 Lamport 的工作，包括那篇尚未发表的 Paxos 论文 (当时还是技术报告的形式)。Lampson 的摇旗呐喊，直接导致了 Lamport 论文的正式发表 (与 8 年前第一次投稿相比，几乎未改)。

这之后，涌现出了一系列关于 Paxos 的研究文献。1999 年，Roberto De Prisco、Butler Lampson 以及 Nancy Lynch （又一个分布式计算领域的大牛）在理论计算机科学 (Theoretical Computer Science) 发表了一篇文章 [PLL97]，重新描述和证明了 Paxos 算法，并分析了其时间开销及容错性。Liskov 等人在 1999 年提出了 PBFT(实用的拜占庭容错算法)[MC99]，这实际上也是 Paxos 的一个变种，被 Lampson 称为 Byzantine Paxos，该算法对基础 Paxos 进行了改进，使其可以处理拜占庭错误。

Eli Gafni 和 Lamport 在 2000 年提出了 Disk Paxos[GL03]，这可以认为是 Paxos 基于磁盘的版本，以支持持久化。随后，Lamport 在 2004 年和 2006 年分别提出了 Cheap Paxos [LM04] 和 Fast Paxos [Lam06]，Cheap Paxos 提升了算法的容错性，而 Fast Paxos 则降低了消息延迟。2007 年，谷歌公司研究小组所提出的 Multi-Paxos [CGR07] 则将基础 Paxos 中 2 阶段简化为 1 阶段，提高了效率。2001 年，Lampson 的论文“The ABCD's of Paxos”[Lam01]，对当时已有的 Paxos 进行了梳

理，分为四类：Abstract Paxos，Classicla Paxos，Byzantine Paxos，以及 Disk Paxos。通过以上描述，我们可以看出，Paxos 已经呈现出百花齐放的趋势，并且已经引起了工业界的注意。

第三个阶段，硕果累累。本阶段，Paxos 开始在工业界得到了广泛应用。这一阶段可以认为是从 2006 年开始的。在那一年，谷歌公司有两篇影响深远的论文发表在 OSDI 上，一篇是“Bigtable:A Distributed Storage System for Structured Data”，另一篇“The Chubby lock service for loosely-coupled distributed systems”。在第二篇论文中，特别提到“Indeed, all working protocols for asynchronous consensus we have so far encountered have Paxos at their core”(事实上，到目前为止我们所遇到的解决异步共识问题的实用算法，其核心都是 Paxos)。

上述两篇论文可以说是揭开了大数据管理的序幕，而 Paxos 则在大数据管理的核心技术 (容错) 中扮演了极为重要的角色。在这之后，Paxos 逐渐被工程技术人员了解，在工业界得到越来越多的应用。在下一节，我们将具体介绍 Paxos 在工业界的应用情况。

## 典型应用场景

下面主要介绍 Paxos 常用的应用场合，我们会简单介绍一个应用方向，然后介绍典型的实例。

### 数据库备份

Paxos 最常见的应用场景是数据库备份 (Database Replication)，保证数据在多个节点上的一致性。工业界在这方面的典型系统包括 Chubby (谷歌公司)，ZooKeeper (Yahoo! 的 Hadoop 项目)，Nutanix (Vmware)，以及 PhxPaxos (腾讯微信)。

Chubby 使用 paxos 来保证日志在各个副本上的一致性，Paxos 算法可以确保每个副本的本地日志具有相同的内容，在这之上是高容错的分布式数据库层。Chubby 被应用在谷歌的多个项目中，包括 GFS 和 Bigtable。

ZooKeeper 可以看作是一个开源的 Chubby 实现，其设计思想类似于 Paxos，但有细微差别。基于 Zookeeper，Hadoop 可以提供强一致性保证的分布式文件系统。ZooKeeper 在 Yahoo! 内部已经成功应用在多个项目中，包括 HBase 及 Yahoo! Message。

Vmware 开发的 Nutanix 分布式文件系统 (NDFS) 是其虚拟计算平台的核心，该系统负责管理所有的元数据和数据。NDFS 具有极高的容错能力，可确保节点发生故障时数据的可用性和一致性。该系统采用 Paxos 来保证数据的强一致性，并采用了仲裁 (Quorum，本书第 6 章) 来进行领导节点的选举。

PhxPaxos 是腾讯公司微信后台团队自主研发的一个类库，基于 Paxos 算法思想，实现多机的状态拷贝。基于 PhxPaxos，可以方便地实现将单机状态扩展到多机，达到容错的强一致性多机状态复制的目的。该类库已在腾讯内部，特别是微信系统中得到了严格的工程验证，目前已经开源。

**Name Server**

网络中每个节点都有一个地址，网络能根据消息的目标地址将消息准确送到对应的节点。域名服务器 (Name Server) 的作用就是将一个节点或服务的名称转换为对应的位置或地址。一个中心域名服务器必须位于一个众所周知的地址，且永不改变。但当这个中心域名服务器崩溃时，整个网络都将崩溃。一个中心域名服务器也需要一个极多的存储空间，并且可能导致消息过载。为了解决上述问题，我们可以采用分布

式域名服务器。

通过 Paxos 算法来管理域名服务，则可保证系统的高可用性，将可用的服务分配给合适的客户端。

**Config 配置管理**

通常对于小的系统，我们习惯采用手工修改配置文件的方法，这样做有两个问题，其一是容易出错；其二，若系统运行在多个节点上，手工修改难以保证多个节点的状态是一致的。因此，对于大规模的应用系统，特别是分布式的应用，我们必须采用自动化的方式来统一修改配置文件。目前一个流行的做法是采用 Zookeeper (核心是 Paxos)，将配置文件放到 Zookeeper 的某个目录上，然后各个程序对这个目录节点进行监听。若配置发生改变，各个节点上的应用程序就会收到通知，并自行修改配置。

# 参考文献

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: An engineering perspective. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 398–407, New York, NY, USA, 2007. ACM.

[GL03] Eli Gafni and Leslie Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[Lam96] Butler W. Lampson. *How to build a highly available system using consensus*, pages 1–17. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Butler Lampson. The abcd's of paxos. In *Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing*, PODC '01, pages 13–, New York, NY, USA, 2001. ACM.

[Lam06] Leslie Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[LM04] Leslie Lamport and Mike Massa. Cheap paxos. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 307–314, 2004.

[MC99] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[OL88] Brian M. Oki and Barbara H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, PODC '88, pages 8–17, New York, NY, USA, 1988. ACM.

[PLL97] Roberto De Prisco, Butler W. Lampson, and Nancy A. Lynch. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms*, WDAG '97, pages 111–125, London, UK, UK, 1997. Springer-Verlag.

# 第 3 章　共识机制

## 3.1　两个朋友约饭局

爱丽丝和鲍勃想要约一个饭局，不过他们又不愿意打电话约，爱丽丝就给鲍勃发了一个短信，想要约在晚上六点。但是，短信不太可靠，有可能在发送过程中丢失，爱丽丝也就没有办法保证鲍勃收到了她的短信，因此她只有收到鲍勃的确认短信后才会去赴约。但是对鲍勃来说，他也不能保证爱丽丝收到了他回复的确认短信。如果确认短信传输途中丢失了，她也不能判断出是鲍勃根本没有收到她的约会短信，还是她没有接收到鲍勃的确认短信。因此，鲍勃可能要求爱丽丝给他发一个确认短信，由此确认她收到了他的回复并且会准时赴约。但是，这条短信发送过程中也可能会丢失……如果爱丽丝和鲍勃两个人都要明确对方能准时赴约，这样的短信确认过程有可能永远持续下去！

**评论:**

- 这样的协议是不能终止的：假设存在能达成意见一致的协议，并且 $P$ 是其中需要消息数量最少的一种协议。因为最后一条确认消息有可能丢失，但是协议仍然要求确保达成一致，我们简单地认为总是可以忽略最后一条消息。这样就产生一个新的协议 $P'$，这个 $P'$ 需要的消息数量比 $P$ 更少，这和假设的 $P$ 需要最少数量消息的假设矛盾。
- 爱丽丝和鲍勃约饭局可以使用 Paxos 算法吗？

## 3.2 共识

在第 2 章中我们学习了含糊地称之为“协定”(Agreement) 的问题。我们现在介绍这个问题的正式定义，称为“共识”(Consensus)。

**定义** 3.1 (**共识** (Consensus)). 系统中有 $n$ 个节点，其中最多有 $f$ 个节点可能崩溃，也就是说，最少有 $n-f$ 个节点是好的。节点 $i$ 从一个输入值 $v_i$ 开始。所有节点必须要从全部输入值中最终选择一个值 (决策值)，并且满足下面的条件。

- **一致性 (Agreement)：** 所有好节点的决策值必定相同。
- **可终止性 (Termination)：** 所有好节点在有限的时间内结束决策过程。
- **有效性 (Validity)：** 选择出的决策值必须是某个节点的输入值。

**评论:**

- 我们假设每一个节点都可以向其他任何节点发送消息，并且节点间的消息链接是可靠的，也就是说，发送的消息一定会到达接收方。
- 不存在广播渠道。如果一个节点要给多个节点发送同一条消息，则必须给这些节点都单独发送一遍。
- Paxos 算法满足这三个要求吗？如果你仔细研究 Paxos 算法，你会注意到 Paxos 算法并不能保证可终止性。例如，如果两个客户端持续地请求票，并且哪一个都不能获得过半数服务器的票，那么系统就会永远卡在那儿。

## 3.3 共识的不可能性

**模型** 3.2 (**异步** (Asynchronous)). 在异步模型 (Asynchronous Model) 中，算法是基于事件的 (“一旦收到消息……，则开始执行……”)。节点不能访问同步时钟。从一个节点发送的消息到达另一个节点的时间是有限的，但没有具体的上限。

**评论:**

- 异步时间模型广泛应用于可变消息延迟模型 (模型 2.6)

**定义** 3.3 (**异步运行时** (Asynchronous Runtime)). 对于异步模型中的算法，运行时 (Runtime) 是指在最坏情况下 (任意合法输入和任意执行场景)，从开始执行到运行结束时所需的时间单位数量。我们假设每个消息最多有一个时间单位的延迟。

**评论:**

- 最大延迟不能用于算法设计，也就是说，算法必须不依赖于真实延迟而工作。
- 异步算法可以被视为一个系统，系统中的本地计算显著快于消息延迟，这样可以认为本地计算不需要时间。节点只有当一个事件发生时 (例如一条消息到达) 才激活，并且“立刻”执行所需要的动作。
- 我们会证明在异步模型中的节点崩溃错误是非常致命的。特别地，异步模型下不存在确定性容错共识算法 (Deterministic Fault-Tolerant Consensus Algorithm)，即使对布尔型输入值也是如此。

**定义** 3.4 (**配置** (Configuration)). 我们认为 (在执行过程中的任何时间点) 一个系统可以通过它的一个配置 (Configuration) $C$ 来完全定义。配置包括了每一个节点的状态，所有正在传输的消息 (即已经发送但还未被接收的消息)。

**定义** 3.5 (**单价** (Univalent)). 我们称一个配置 $C$ 是单价 (Univalent) 的，如果决策值不依赖于之后发生的事情就可以确定。

**评论:**

- 若一个配置为单价的且相应的决策值为 $v$。我们称这样的配置为 $v$- 价 ($v$-valent)。
- 一个配置可以是单价的，甚至没有一个节点意识到这一点。例如，一个配置中的所有节点的输入值为 0，则这个配置是 *0*-价[1]。
- 当输入值被限定为布尔值时，任何共识算法的决策值也会是布尔值 (同样是依据有效性要求)。

**定义** 3.6 (**二价** (Bivalent)). 若节点可以决策的值只能是 0 或者 1，则称这个配置 $C$ 为二价 (Bivalent) 的。

**评论:**

- 决策值依赖于消息接收的顺序或者崩溃事件发生的顺序，也就是说，那时尚未决定选择哪个值。
- 我们把一个算法的初始配置称为 $C_0$。当节点处于 $C_0$ 配置时，所有节点都执行初始化代码，也可能发送一些消息，并且等待第一个消息到达。

[1]译者注：根据有效性要求，决策值必须是某个节点的输入值，那么此时决策值只能是 0。

**引理** 3.7. 若 $f \geqslant 1$，至少存在一组输入值 $V$，使得相应的初始配置 $C_0$ 是二价的。

证明. 注意到 $C_0$ 仅仅由节点的输入值决定，因为还没有任何事件发生。我们用 $V = [v_0, v_1, \ldots, v_{n-1}]$ 来描述一组输入值，这里 $v_i$ 是节点 $i$ 的输入值。

我们可以构造 $n+1$ 个数组 $V_0, V_1, \ldots, V_n$，这里下标 $i$ 表示数组 $V_i$ 中的一个特殊位置：$V_i$ 的第 0 到第 $i$ 个元素都为 1，而其余元素均为 0。因此，$V_0 = [0, 0, 0, \ldots, 0]$，$V_1 = [1, 0, 0, \ldots, 0]$，依此类推，直到 $V_n = [1, 1, 1, \ldots, 1]$。

为了满足有效性，与 $V_0$ 对应的*初始配置*必然是 0-价的[2]，类似地，与 $V_n$ 对应的*初始配置*必然是 1-价的。假设与每个 $V_i$ 对应的*初始配置*均为单价的。那么，必然存在至少一个下标 $b$，使得与 $V_b$ 对应的*初始配置*是 0-价的，而与 $V_{b+1}$ 对应的*初始配置*是 1-价的[3]。注意这里，$V_b$ 和 $V_{b+1}$ 的唯一区别是第 $b$ 个节点的输入值。

由于我们假定算法可以容忍至少一个错误节点，即 $f \geqslant 1$，我们考虑下面的执行场景：网络的输入值分别为 $V_b$ 和 $V_{b+1}$[4]。节点 $b$ 非常慢，从它发出的所有消息在网络上需要传输很长的时间，以至于其他节点不得不假定节点 $b$ 崩溃了，否则就会违背*可终止性*的要求。由于其他节点不知道节点 $b$ 的输入值，并且我们假定所有的*初始配置*都是单价的，(除 $b$ 之外) 的节点们将选择一个决策值 $v$，而这个决策值与节点 $b$ 的值无关。由于 $V_b$ 是 0-价的，$v = 0$；而 $V_{b+1}$ 是 1-价的，$v = 1$。这显然是

[2]译者注：即决策值必定为 0。

[3]译者注：注意 $V_n$ 是 1-价的，若 $V_{n-1}$ 是 0-价，则 $b = n-1$；否则，继续检测 $V_{n-1}, \cdots, V_0$，遇到的第一个 0-价的 $V_k$，就会得到 $V_k$ 为 0-价，而 $V_{k+1}$ 为 1-价，让 $b = k$ 即可。因为 $V_0$ 是 0-价，这样的 $k$ 必定存在。

[4]译者注：实际上只有第 $b$ 个节点的输入值不同。

矛盾的。

**定义** 3.8 (**转换** (Transition)). 转换 (Transition)：从配置 $C$ 到下一个配置 $C_\tau$ 的转换由事件 $\tau = (u, m)$ 标志，即“节点 $u$ 接收到消息 $m$”。

**评论:**

- 转换是对之前描述的异步模型中“事件”的正式定义。
- 如果消息 $m$ 在配置 $C$ 中仍然处于传输状态，相应的转换 $\tau = (u, m)$ 只可应用于配置 $C$。
- $C_\tau$ 和 $C$ 存在以下区别：消息 $m$ 不再处于传输状态；节点 $u$ 可能有一个不同的状态 (因为 $u$ 可以基于 $m$ 更新自己的状态)，并且可能会有从 $u$ 发出的新消息正处于传输状态中。

**定义** 3.9 (**配置树** (Configuration Tree)). 配置树是多个配置组成的有向树。树的根是 $C_0$，$C_0$ 完全由输入值集合 $V$ 决定。树的边是转换，每个配置都是一个节点，其所有可应用的转换都是节点的出边。

**评论:**

- 对任何一个算法，每一组输入值都只有唯一的配置树。
- 算法执行结束时的配置是树的叶节点。要注意的是，我们说的“结束”是系统作为一个整体都结束了，也就是说，系统不会再做任何转换。
- 从根节点到叶子节点的每一条路径都是一个可能的算法异步执行过程。
- 叶子节点必须是单价的，否则会导致算法没有达成协定就结束。

- 当系统处于配置 $C$ 时，如果此时一个节点 $u$ 崩溃，在配置树中所有和 $u$ 相关的转换 $(u,*)$ 都会从 $C$ 的出边中删除。

**引理** 3.10. 假设两个转换 $\tau_1 = (u_1, m_1)$ 和 $\tau_2 = (u_2, m_2)$ $(u_1 \neq u_2)$ 都可应用于配置 $C$。若 $C_{\tau_1\tau_2}$ 是在 $C$ 上先应用 $\tau_1$ 再应用 $\tau_2$ 得到的配置，且 $C_{\tau_2\tau_1}$ 也用类似的方式定义。则 $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$。

证明. 由于 $m_2$ 仍然处于传输状态，并且 $\tau_1$ 不能改变 $u_2$ 的状态，因此 $\tau_2$ 是可以应用在 $C_{\tau_1}$ 上的。同理，$\tau_1$ 可以应用在 $C_{\tau_2}$ 上。这样 $C_{\tau_1\tau_2}$ 和 $C_{\tau_2\tau_1}$ 就都定义清楚了。因为这两个转换是完全彼此独立的，这意味着它们消耗了同样的消息，并导致同样的状态转换，以及同样的待传输消息。于是就可以推导出 $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$。

**定义** 3.11 (**关键配置** (Critical Configuration)). 一个配置 $C$ 是关键的，如果 $C$ 是二价的，但是配置树中 $C$ 的所有直接子节点的配置都是单价的。

**评论:**

- 通俗地讲，如果配置 $C$ 是关键的，就表明这是执行过程中“决策值尚未确定”的最后时刻。一旦下一个消息被任何节点处理，则决策值会被确定。

**引理** 3.12. 如果一个系统是在一个二价的配置中，这个系统必须在有限的时间内达到一个关键配置，否则这个系统就永远不能解决共识问题。

证明. 在引理 3.7 中我们已经证明了至少有一个二价的初始化配置。假定这个配置不是关键配置，则其子节点中必然至少有一个二价配置，那么系统有可能进入这个二价配置。但是如果这个配置也不是关键配置，

系统则可能后继进入到另一个二价配置。只要不存在关键配置，总存在一种执行选择使得系统进入到另一个二价配置中。要确保一个算法到达一个单价配置的唯一方式就是先到达一个关键配置。

因此我们能够断定，一个没有到达关键配置的系统至少有一个可能的执行路径，使得系统在一个二价配置中结束 (此时算法结束，但没能达成一致)，否则根本就不会结束。

**引理** 3.13. 如果一个配置树包含一个关键配置，单个节点的崩溃会导致一个二价的叶子节点，也就是说，单节点崩溃阻碍了算法达成一致。

证明. 假定 $C$ 是一个配置树中的关键配置，我们用 $T$ 表示可应用于 $C$ 的转换的集合。现在，$\tau_0 = (u_0, m_0) \in T$ 以及 $\tau_1 = (u_1, m_1) \in T$ 是两个转换，并且让 $C_{\tau_0}$ 是 0-价[5] 的，而 $C_{\tau_1}$ 是 1-价的。须注意 $T$ 必然包含这些转换，因为 $C$ 是一个关键配置。

假设 $u_0 \neq u_1$。根据引理 3.10，我们知道 $C$ 有一个后继配置 $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$。既然这个配置是在 $C_{\tau_0}$ 之后，则它一定是 0-价的[6]。但是这个配置也在 $C_{\tau_1}$ 之后，所以它一定也是 1-价的。这个茅盾推翻了原假设，因此我们可以断定 $u_0 = u_1$ 必然成立。

因此我们可以选择一个节点 $u$,使得可以找到一个转换 $\tau = (u, m) \in T$，该转换可以从 $C$ 到达一个 0-价的配置。如之前所示，$T$ 中所有运行至一个 1-价配置的转换都必须涉及节点 $u$。既然 $C$ 是关键的，则必定存在至少一个这样的转换 $\tau$。同理，可得出 $T$ 中所有运行至一个 0-价配置的转换也必须涉及节点 $u$。并且因为 $C$ 是关键的，所以 $T$ 中不

[5]译者注：决策值为 0。

[6]译者注：一个 $v$-价的配置，其所有后继配置必然也是 $v$-价的。也就是说，决策值已经确定。

存在转换可以到达一个二价配置。因此所有可应用于 $C$ 的转换都要经过同一个节点 $u$!

如果系统处于配置 $C$ 时节点 $u$ 崩溃，所有的转换都会被删除，因此系统会僵持在配置 $C$ 中，也就是说在 $C$ 结束。但是因为 $C$ 是关键的，因此也就是二价的，所以算法不能达成一致。

**定理** 3.14. 当 $f > 0$ 时，不存在一个确定的算法 (Deterministic Algorithm) 总能在异步模型下达成共识。

证明. 我们假设输入值是布尔值，这也是最容易的一种可能性。从引理 3.7 中我们知道存在至少一个二价的初始配置 $C$。利用引理 3.12 我们知道，如果一个算法能解决共识问题，从二价初始配置 $C$ 开始的所有执行都必须到达一个关键配置。但是如果算法到达一个关键配置，则存在一个单节点崩溃使得算法不能达成共识 (引理 3.13)。

**评论:**

- 如果 $f = 0$，则每一个节点都只需要把自己的值发送给所有其他节点，然后等待所有的值，并从中选取一个最小值。
- 但是如果某个节点崩溃，则没有一个确定的方案能在异步模型中达成共识。
- 这种情况如何改进？例如，通过让每个节点都能访问随机值，也就是说，允许每个节点以抛硬币的方式选取一个值。

## 3.4 随机共识

**算法 3.15** 随机共识 (Ben-Or)

```
1: v_i ∈ {0,1}                    ◁ 输入值
2: round = 1
3: decided = false
4: 广播 myValue(v_i, round)
5: while true do
       提案
6:     持续等待，直到收到当前轮 (round) 过半数的 myValue 消息
7:     if 所有消息包含相同的值 v then
8:         广播 propose(v, round)
9:     else
10:        广播 propose(⊥, round)
11:    end if
12:    if decided then
13:        广播 myValue(v_i, round+1)
14:        确定决策值为 v_i 并且终止
15:    end if
       调整
16:    持续等待，直到收到过半数当前轮 (round) 的 propose 消息
17:    if 所有消息提案 (propose) 同样的值 v then
18:        v_i = v
19:        decide = true
20:    else if 至少有一个提案包含 v then
21:        v_i = v
22:    else
23:        随机选择 v_i 的值，Pr[v_i = 0] = Pr[v_i = 1] = 1/2
24:    end if
25:    round = round + 1
26:    广播 myValue(v_i, round)
27: end while
```

**评论:**

- 算法 3.15 的思路非常简单：要么所有的节点都从相同的输入值开始，这很容易达成共识。否则，所有的节点都抛硬币，直到其中的大部分节点碰巧得到一个相同的值。

**引理** 3.16. 只要还没有节点将变量 *decided* 设置为 true，算法 3.15 将一直运行下去，和哪个节点崩溃并没有关系。

证明. 上述算法中，节点只有在第 6 行和第 16 行[7]处于等待状态。因为一个节点只需等待过半数其他节点发送信息，而且因为 $f < n/2$，只要还没有一个好节点将变量“decided”设置为“true”并且结束运行，节点最后总会接收到足够多的消息以继续运行。

**引理** 3.17. 算法 3.15 满足有效性要求。

证明. 注意到当要求输入值为布尔值时，共识的有效性要求对应于：如果所有的节点的输入值均为 $v$，那 $v$ 必须是最后的决策值；否则，只有 0 或 1 是可接受的，这样的话有效性要求就自动满足了。

假设所有的节点的输入值均为 $v$，在这种情况下，所有的节点在第一轮中就开始提议 $v$ 为决策值。既然所有的节点只能收到提议 $v$ 的消息，所有的节点也会决定将 $v$ 作为决策值 (第 17 行)，并且在下一轮中退出循环。

**引理** 3.18. 算法 3.15 满足一致性要求。

证明. 节点只在接收到过半数包含 $v$ 的消息时 (如算法第 8 行所示)，才发送推举 $v$ 的提案，因此在同一轮中不可能同时出现关于 0 和 1 的提案。

[7]译者注：原书为 15 行，有误。

假设 $u$ 是在第 $r$ 轮中第一个决定决策值是 $v$ 的节点，这样 $u$ 必然在第 $r$ 轮中接收到了过半数推举 $v$ 的提案 (第 17 行)。须注意：一旦一个节点接收到过半数推举同一个值的提案，这个节点就会接受这个值 (将本地的 $v_i$ 修改为 $v$) 并且在下一轮中结束运行。既然在 $r$ 轮中不可能有推举其他值的提案，可以推理出在 $r$ 轮中不会有节点选定另一个不同的值。

在引理 3.16 中，我们只是表明了只要还没有节点确定了决策值，节点就会持续运行，因此我们现在需要留意如果节点 $u$ 结束时其他节点不会被卡住。

任何一个节点 $u' \neq u$ 可能经历以下两种场景：要么这个节点在第 $r$ 轮中接收到过半数推举 v 的提案，并且决定采纳 $v$；要么这个节点没有收到过半数一致的提案。在第一种情况中，达成共识的要求可以直接满足，并且节点也不可能卡住。我们现在只需要研究后一种情况。因为节点 $u$ 接收到大部分推举 $v$ 的提案，这说明每一个节点都接收到 至少一个推举 $v$ 的提案[8]。

因此，所有的节点在 $r$ 轮都把它们各自的值 $v_i$ 设置为 $v$(算法 17、18 行)。这样的话，所有的节点在第 $r$ 轮结束时都会广播 $v$，因此在第 $r+1$ 轮时所有的节点都会提议 $v$。在第 $r$ 轮中就已经决定采纳 $v$ 的节点在 $r+1$ 轮中会结束运行，并且发送一个额外的 `myValue` 消息 (第 13 行)。

所有的其他节点在第 $r+1$ 轮中均会收到过半数推举 $v$ 的提案，并且在此轮决定采纳 $v$，而且也同样发送一个 `myValue` 消息。这样，在第 $r+2$ 轮时，有一些节点已经结束运行了，剩余的其他节点接收到足够的 `myValue` 消息 (第 6 行)。这些节点发送一条 `propose` 和一条 `myValue`

[8]译者注：根据算法第 16 行，每个节点必须接收到过半数节点的消息才结束等待。显然这过半数消息中必然至少一个是推举 $v$ 的提案。

消息，并且在第 $r+2$ 轮中决定采纳 $v$，结束运行。

**引理** 3.19. 算法 3.15 满足可终止性要求, 且所有节点终止时间的期望为 $O(2^n)$。

证明. 我们从引理 3.18 的证明中已经知道，一旦一个节点接收到过半数推举 $v$ 的提案，所有的节点都将在最多两轮后结束。因此，我们只需要证明一个节点收到过半数推举同一个值的提案的期望时间为 $O(2^n)$。

假定没有一个节点接收到推举同一个值的过半数提案。在这样的一轮中，某些节点可能基于某个收到的提案内容把它们的值更新为 $v$ (第 20 行)。如前所述，所有基于某个提案内容更新它们本地值的节点都会采纳同样的 $v$[9]。

其他的节点随机地选择 0 或者 1。因此，所有节点在一个轮次中选定相同值 $v$ 的概率至少为 $1/2^n$。因此，预期的轮数是由 $O(2^n)$ 界定的。因为每一轮包含了两次消息交换，则算法的运行时间大致和轮次相同。

**定理** 3.20. 算法 3.15 能在最多 $f < n/2$ 个节点崩溃的情况下在期望时间 $O(2^n)$ 内达成布尔型的共识。

**评论:**

- $f < n/2$ 的容错能力到底有多好?

**定理** 3.21. 不存在一个异步模型的共识算法能容忍 $f \geqslant n/2$ 个坏节点。

[9]译者注：见引理 3.18 的证明，同一轮中的提案必然包含相同的值。

证明. 假设存在一个算法能够处理 $f = n/2$ 个错误的节点。我们可以把所有节点分成两个集合 $N$ 和 $N'$，分别包含 $n/2$ 个节点。我们可观察三组不同的输入值：$V_0$: 所有节点输入值为 0; $V_1$: 所有节点输入值为 1; $V_{\text{half}}$: 所有 $N$ 中的节点输入值为 0，而所有 $N'$ 中的节点输入值为 1。

假定节点输入值为 $V_{\text{half}}$。因为算法必须独立于消息的传输情况来解决共识问题，我们可以研究这样的场景：所有从 $N$ 中的节点发往 $N'$ 中 (或者反过来) 的消息都严重延迟。注意到 $N$ 中的节点不能断定它们的输入值是 $V_0$ 还是 $V_{\text{half}}$。类似地，$N'$ 中的节点也不能断定它们的输入值是 $V_1$ 还是 $V_{\text{half}}$。因此，如果从 $N$ 发向 $N'$ 的任何一条消息到达之前，算法就已结束，则 $N$ 必然决定选取决策值 0，而 $N'$ 必然决定选取决策值 1 (为了满足有效性要求)。这样，算法不能达成共识。

解决这个问题的唯一可能就是等待至少一条从一个子集发到另一个子集的消息到达目的地。但是，因为 $f = n/2$ 个节点可能崩溃，整个子集中的节点有可能在发出消息前就全部崩溃了。在这种情况下，算法会一直等待下去，也就不能满足可结束性要求。

**评论:**

- 算法 3.15 解决了共识问题且具备最强的容错能力[10]，但是性能非常非常慢。这个问题的根源在于每个节点独立地掷硬币方法：如果所有的节点都掷同一个硬币，那么就能在固定轮数中结束算法。
- 这个问题可以通过简单地总是选择 1 (程序 22 行) 来解决吗？
- 这是行不通的：这样的改变会使得算法变为了确定性算法 (Deterministic)，因此也就不可能达成共识 (定理 3.14)。通过模拟

[10]译者注：上面的定理已说明不可能容忍更多的错误节点。

总是选择 1，会发现可能发生以下情况：过半数节点选择 0，但是少数选择 1 的节点阻碍了算法达成最终的共识。

- 尽管如此，算法还是可以通过掷一个称为*共享硬币*的方式来改进。*共享硬币*是一个随机变量，对所有节点以恒定的概率产生 0 或者 1 值。当然，这样的硬币也不是魔法设备，就是一个简单的算法而已。为了改进算法 3.15 的运行效率，我们把算法的第 23 行改为调用共享硬币算法的函数。

## 3.5 共享硬币

**算法 3.22** 共享硬币 (在节点 $u$ 上运行的代码)

1: 以 $1/n$ 的概率使本地硬币 $c_u = 0$，否则 $c_u = 1$
2: 广播 myCoin($c_u$)
3: 等待 $n - f$ 个硬币到来，并将它们存储在本地的硬币集合 $C_u$
4: 广播 mySet($C_u$)
5: 等待 $n - f$ 个硬币集合
6: **if** 在这些集合中，至少有一个硬币为 0 **then**
7: return 0
8: **else**
9: return 1
10: **end if**

**评论:**

- 既然最多有 $f$ 个节点崩溃，所有的节点总会接收到 $n - f$ 个硬币和 $n - f$ 个硬币集 (第 3 行和第 5 行)。因此，所有的节点能够顺利运行并且也能保证结束。

- 我们现在证明算法在 $f < n/3$ 时的正确性。为了简化证明过程，我们假定 $n = 3f + 1$，也就是说，我们假定最坏的情况。

**引理** 3.23. $u$ 是一个节点，$W$ 是硬币集，$W$ 中的每个硬币至少出现在 $f + 1$ 个被 $u$ 接收到的硬币集合中。则 $|W| \geqslant f + 1$ 成立。

证明. 我们用 $C$ 来标记 $u$ 接收到的所有硬币的多重集合[11]。注意，$u$ 接收到的硬币正好是 $|C| = (n - f)^2$ 个 (可能有重复)。因为 $u$ 接收了 $n - f$ 个硬币集，而每一个硬币集中包含 $n - f$ 个硬币。

假设引理 3.23不成立，那么，最多只有 $f$ 个硬币出现在全部 $n - f$ 个硬币集中，而其他所有的 $(n - f)$ 个硬币最多只出现在 $f$ 个硬币集中。换句话说，$u$ 接收到的所有硬币数将受下面的限制：

$$|C| \leqslant f \cdot (n - f) + (n - f) \cdot f = 2f(n - f).$$

12

我们的假设是 $n > 3f$，也就是说，$n–f > 2f$。因此，$|C| \leqslant 2f(n–f) < (n - f)^2 = |C|$: 矛盾。

**引理** 3.24. 所有好节点都能看到 $W$ 中的所有硬币。

证明. 设硬币 $w \in W$。根据 $W$ 的定义，我们知道 $w$ 至少出现在 $f+1$ 个节点 $u$ 所接收到的硬币集中。由于每一个其他节点也需要接收到 $n - f$ 个硬币集才能结束等待，因此每一个节点也将收到上述 $f + 1$ 个硬币集合中的至少一个。由此可见，$w$ 必然发送到在每一个节点中。

[11]译者注：多重集合中，同一个元素可以出现多次。

[12]译者注：若引理 3.23不成立，则最多有 $f$ 个硬币出现在 $f+1$ 个集合中。但是 $n = 3f+1$，因此 $f + 1 < n - f$。这样我们依旧得到了上述式子。

**定理** 3.25. 如果崩溃的节点最多为 $f < n/3$, 算法 3.22 实现了一个共享的硬币[13]。

证明. 我们先估计算法中所有节点都返回 1 的概率值的下界。根据算法第 1 行，所有节点选择硬币值等于 1 的概率为 $(1-1/n)^n \approx 1/e \approx 0.37$。在这种情况下 1 会成为最终的决策值。这只是所有节点选择 1 为决策值的下限，在很多其他情况下，1 也有机会成为决策值，这取决于消息传输的情况以及节点崩溃的情况。但是 0.37 的概率已经足够好了，所以我们也不需要再考虑这些其他的场景。

在 $W$ 中至少出现一个 0 的概率为 $1-(1-1/n)^{|W|}$，根据引理 3.23 我们知道 $|W| \geqslant f+1 \approx n/3$，因此这个概率值约为 $1-(1-1/n)^{n/3} \approx 1-(1/e)^{1/3} \approx 0.28$。这个 0 将被所有的节点看到 (引理 3.24)，因此所有节点都决定选择 0 值。

由此可见，算法 3.22 实现了一个共享的硬币。

**评论:**

- 我们只证明了最坏的情况。通过选择较小的 $f$，很显然 $f+1 \not\approx n/3$。但是，引理 3.23 也适用于 $|W| \geqslant n-2f$ 的情况。要证明这个结论需要替换在证明过程中导出矛盾的论述：最多 $n-2f-1$ 个硬币能出现在所有的 $n-f$ 个硬币集中，并且 $n-(n-2f-1)=2f+1$ 个硬币可出现在最多 $f$ 个硬币集中。剩下的证明过程就类似了，唯一的不同在于其中的数学部分不是很简单，需要稍作变换。利用修改的引理我们知道 $|W| \geqslant n/3$，因此定理 3.25 在任何 $f < n/3$ 的情况下都成立。

[13]译者注：指所有节点决策 1 或 0 的机会大致相同。

我们隐含了一个假设：消息的传递时间和顺序是随机的。如果我们需要一个 0 值，但是想要推举 0 值的节点发送消息很缓慢，那么没有节点会看到这些 0 值，这样的话算法也不能继续进行。

**定理** 3.26. 将算法 3.22 和算法 3.15 结合起来，我们就得到一个随机共识算法，这个算法能在常数级的可预期的轮数中结束，并且能容忍最多 $f < n/3$ 个节点崩溃。

## 章节说明

两个朋友安排会面的问题已经有多种不同的名称来阐述和研究了，如今，这个问题通常被称为“两将军问题”。不可能性的证明发表在 1975 年的 Akkoyunlu 等人的论文中 [AEH75]。

对不存在一个确定性算法总是能解决共识问题的证明是以 1985 年 Fischer, Lynch 和 Paterson [FLP85] 的证明为基础的，称为 FLP。这个结果获得了 2001 年的 PODC 最有影响论文奖 (PODC Influential Paper Award，现在称为 Dijkstra Prize)。随机共识算法的思想源于 Ben-Or [Ben83] 提出的思想。共享硬币的概念是 Bracha [Bra87] 提出的。

## 参考文献

[AEH75] EA Akkoyunlu, K Ekanadham, and RV Huber. Some constraints and tradeoffs in the design of network communications. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 67–74. ACM, 1975.

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In

*Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

# 第 4 章　拜占庭协定

为了提高飞行的安全系数，学者们曾研究过安装在飞机上数量众多的传感器和仪器可能发生的错误。在对这些错误进行建模的过程中，他们遇到了一个问题：失灵的仪器不但会停止工作，有时还呈现出任意的行为。基于这样的认知，学者们认为仪器错误可以是任何类型的，不局限于任何模式。

**定义** 4.1 (**拜占庭**). 一个可能呈现任意行为的节点被称为拜占庭。任意行为意味着“所有能想象到的事情”，比如，根本不发送任何消息，向不同的邻居发送不同且错误的消息，以及谎报自己的输入值。

**评论:**

- 拜占庭行为也包含串谋，即所有的拜占庭节点被同一个攻击者控制。
- 我们假定任何两个节点之间直接通信，并且没有一个节点可以伪造其他节点的发送地址。这个要求确保了单个拜占庭节点不能扮演所有的节点。
- 我们将所有非拜占庭的节点称为好节点。

**定义** 4.2 (**拜占庭协定** (Byzantine Agreement)). 在一个存在拜占庭节点的系统中达成如定义 3.1 所述的共识被称为拜占庭协定。如果一个算法可以在存在 $f$ 个拜占庭节点的情况下正确工作，则称该算法为 $f$-可适用 ($f$-resilient)。

**评论:**

- 与定义 3.1 描述的共识一样，我们也需要一致性，可终止性，以及有效性。一致性和可终止性是显然成立的。但是有效性能被保证吗？[1]

## 4.1 有效性

**定义** 4.3 (**任何输入有效性** (Any-Input Validity)). 最终的决策值必须是某个节点的输入值。

**评论:**

- 在描述定义 3.1的共识时，我们也曾隐含地使用了这个有效性定义。
- 当存在拜占庭节点时，这个定义还有意义吗？如果拜占庭节点谎报它的输入值，会出现什么情况？
- 我们可能希望一个更合理的有效性定义，它能区分拜占庭输入和正确的输入。

**定义** 4.4 (**正确输入有效性** (Correct-Input Validity)). 决策值必须是某个好节点的输入值。

[1]译者注：定义 3.1的有效性指决策值必须是某个节点的输入值。由于拜占庭节点可以伪报自己的输入值，这个要求显然是有疑问的。

**评论:**

- 不幸的是，实现正确输入有效性看起来并不容易，因为一个拜占庭节点可以遵循协议，但是谎报它的输入值，此时无法将其与一个好节点进行区分。于是我们需要一个替代品。

**定义** 4.5 (**全部相同有效性** (All-Same Validity)). 如果所有好节点起始时具有相同的输入值 $v$, 则决策值必须也是 $v$。

**评论:**

- 如果决策值是布尔型的，可以从全部相同有效性得到正确输入有效性。
- 如果输入值不是布尔型的，比如是从传感器采集到的实数值，全部相同有效性在很多场景下并不真正有用。

**定义** 4.6 (**中值有效性** (Median Validity)). 如果输入值是有序的，比如实数值，可以使决策值接近所有正确输入值的中值，以避免拜占庭异常。这里，与中值的差距决定于拜占庭节点的个数 $f$。

**评论:**

- 有可能实现拜占庭协定吗？如果可能，应该采用哪种有效性？
- 让我们尝试设计一个算法，使其可以容忍单个拜占庭节点。我们的讨论首先限制在同步模式 (Synchronous Model)。

**模型** 4.7 (**同步** (Synchronous)). 在同步 (Synchronous) 模式下，节点运行在同步轮次中。在每一轮，每个节点可能向其他节点发送一条消息，接收其他节点发送的消息，并进行某些本地计算。

**定义** 4.8 (**同步运行时** (Synchronous Runtime)). 在同步模式下，运行时 (Runtime) 可以简单地等同于在最坏情况下算法从开始到结束运行了多少轮 (对于任意合法的输入，以及任意执行场景)。

## 4.2 有多少个拜占庭节点

**算法 4.9** 拜占庭协定 ($f = 1$)

1: 在节点 $u$ 上运行的代码，节点的输入值为 $x$:

*第 1 轮*

2: 向所有其他节点发送 $\texttt{tuple}(u, x)$
3: 从所有其他节点 $v$ 接收 $\texttt{tuple}(v, y)$
4: 将所有接收到的消息 $\texttt{tuple}(v, y)$ 存储在集合 $S_u$

*第 2 轮*

5: 向所有其他节点发送 $S_u$
6: 从所有其他节点 $v$ 接收 $S_v$
7: $T =$ 所有出现在至少两个 $S_v$ 中的 $\texttt{tuple}(v, y)$, 包括自己的 $S_u$
8: 设 $\texttt{tuple}(v, y) \in T$ 是 $y$ 值最小的元组
9: 选择 $y$ 为决策值

**评论:**

- 拜占庭节点可能不遵循上述协议并且发送语法上不正确的消息。这样的消息可以很容易地被检测出来并被丢弃。一个糟糕的情况是：如果拜占庭节点发送语法上正确的消息，但是内容是伪造的，例如它们向不同的节点发送不同的内容。
- 某些错误难以被轻易发现。比如，若一个拜占庭节点在第一轮

向不同的节点发送不同的值，这些值将被存入 $S_u$。然而，某些错误可以而且必定能被检测出来。注意到所有的节点在第二轮仅仅转发所接收到的消息，而并不会发送自己的输入值。如果一个拜占庭节点发送一个集合 $S_v$，其中包含了它自己的值 $\texttt{tuple}(v, y)$, 这个元组必然会被接收节点 $u$ 从 $S_v$ 中移除 (算法第 6 行)。

- 由于我们假定节点不能伪造他们的源地址，因此当一个节点在第一轮接收到元组 $\texttt{tuple}(v, y)$ 后，它可以确保该消息是来自于节点 $v$。

**引理** 4.10. 如果 $n \geqslant 4$, 所有好节点都将得到同一个集合 $T$。

证明. 由于 $f = 1$, 且 $n \geqslant 4$, 我们至少有三个好节点。一个好节点将至少两次看到每一个正确的输入值。一次是从对应该输入值的源节点直接得到 (第 1 轮)，另一次是从第三个好节点那里得到 (第 2 轮)。因此所有正确的值都在集合 $T$ 中。如果唯一的那个拜占庭节点将一个相同的值发送给至少两个好节点，那么所有好节点将看到该值两次，并且将它加入到集合 $T$ 中。如果该拜占庭节点发送到其他节点的值两两不同，则这些值都不能进入到集合 $T$ 中。

**定理** 4.11. 如果 $n \geqslant 4$, 算法 4.9 将能达成拜占庭协定。

证明. 我们需要证明一致性、任何输入有效性，以及可终止性。根据引理 4.10，我们知道所有的好节点都将得到同一个集合 $T$，并且都支持同一个最小值。因此节点们将支持同一个值，且这个值是某个节点的输入值。从而任何输入有效性成立。此外，该算法将在两轮之后终止。

**评论:**

- 如果 $n > 4$，(唯一的) 拜占庭节点将能把多个值输入到集合 $T$ 中[2]。
- 算法 4.9 仅仅支持任何输入有效性，而这在拜占庭场景下是有疑问的。如果一个值至少出现两次的话，通过从至少出现两次的值中选择最小值，我们可以得到全部相同有效性。
- 该算法的思想可以被推广到任意 $f$ $(n > 3f)$ 的情况。在推广中，每个节点将它在 $f+1$ 轮所学到的信息发送给其他节点。因此，消息的大小将随 $f$ 的数量呈指数级增长。
- 算法 4.9 适用于 $n = 3$ 的情况吗?

**定理** 4.12. 如果网络只包含 3 个节点，且其中一个为拜占庭节点，则该网络不能达成符合全部相同有效性要求的拜占庭协定。

证明. 设这三个节点为 $u, v, w$。为了达成全部相同有效性，一个好节点若发现另一个节点的值与自己相同，就必须将决策值设定为自己的值[3]。第三个节点可能不同意这个决策值，但是它可能是拜占庭节点。

假设好节点 $u$ 的输入值为 0，而好节点 $v$ 的输入值为 1。拜占庭节点 $w$ 可以愚弄这两个节点：$w$ 告诉 $u$ 它的输入值是 0 (与 $u$ 相同)，同时告诉 $v$ 它的值为 1(与 $v$ 相同)。于是 $u$ 和 $v$ 为了保证全部相同有效性，将选择不同的决策值，这就使得一致性 (定义 3.1) 不能满足。即使 $u$ 和 $v$ 相互通信，并且发现它们所得到的 $w$ 值是不同的，但是 $u$ 也无法区分 $v$ 和 $w$ 到底哪个是拜占庭节点。

[2]译者注：如 $n = 5$, 拜占庭节点可以将两个值分别发送给两个节点，这样它们都能进入 $T$ 中。

[3]译者注：根据全部相同有效性的定义："如果所有好节点起始时具有相同的输入值 $v$, 则决策值必须也是 $v$"。因此当两个"好"节点的值相同时，决策值必须是这个值。

**定理** 4.13. 若一个有 $n$ 个节点的网络中存在 $f \geqslant n/3$ 个拜占庭节点，则该网络不能达成拜占庭协定。

证明. 采用反证法，假定存在一个算法 $A$，可以在一个具有 $n$ 个节点且其中有 $f \geqslant n/3$ 个拜占庭节点的网路中达成拜占庭协定。那么根据算法 $A$，我们可以在一个只有 3 个节点的网络中解决拜占庭问题。为了简单起见，我们将这三个节点称为 $u, v, w$ 超级节点。

每个超级节点模拟 $n/3$ 个节点，如果 $n$ 不能被 3 整除，则可能是 $\lfloor n/3 \rfloor$ 或者 $\lceil n/3 \rceil$ ($n/3$ 的上取整或下取整)。每个被模拟的节点都采用与它对应的超级节点的输入值。现在我们就用这三个超级节点来运行算法 $A$。唯一的拜占庭超级节点代表了 $\lceil n/3 \rceil$ 个拜占庭节点。由于算法 $A$ 保证能在 $f \geqslant n/3$ 的情况下解决拜占庭协定，$A$ 必须能应对 $\lceil n/3 \rceil$ 个拜占庭节点。

算法 $A$ 确保所有被两个好的超级节点所模拟的好节点能达到拜占庭协定。因此这两个好的超级节点可以仅使用在好节点中达成一致的值，于是我们在三个超级节点的网络中达成了一致，而其中一个节点是拜占庭节点。这与定理 4.12 相矛盾，因此算法 $A$ 是不存在的。

## 4.3 国王算法

**引理** 4.15. 算法 4.14 实现了全部相同有效性。

证明. 如果所有好节点初始时拥有相同的输入值，则好节点们在第 2 轮都会提议 (Propose) 这个值。所有好节点将会接收到至少 $n - f$ 个提案 (Proposal)，因此所有好的节点将保持这个值，并且不会切换到国王的值。这个结论对所有阶段都适用。

**算法 4.14** 国王算法 (King Algorithm) ($f < n/3$)

1: $x =$ 本节点的输入值
2: **for** 从第 1 到第 $f+1$ 个阶段 **do**

*第 1 轮*

3: 广播 value($x$)

*第 2 轮*

4: **if** 接收到 value($y$) 至少 $n-f$ 次 **then**
5: 广播 propose($y$)
6: **end if**
7: **if** 接收到 propose($z$) 至少 $f$ 次 **then**
8: $x = z$
9: **end if**

*第 3 轮*

10: 设节点 $v_i$ 是预先确定好的第 $i$ 阶段的国王
11: 国王 $v_i$ 广播它当前的值 $w$
12: **if** 接收到 propose($x$) 的次数严格少于 $n-f$ **then**
13: $x = w$
14: **end if**
15: **end for**

**引理** 4.16. 在 $n > 3f$ 的情况下，如果一个好节点提议 $x$，不会有其他好节点提议另一个值 $y$ ($y \neq x$)。

证明. 采用反证法。假定一个好节点 $A$ 提议 $x$，并且有另一个好节点 $B$ 提议 $y$ ($y \neq x$)。注意到一个好节点只会在接收到 $n-f$ 次包含同一

个值的消息之后才会提议这个值，并且最多只有 $f$ 个拜占庭节点发送 $x$ 给 $A$ 且发生 $y$ 给 $B$。我们可以推出下面的结论：$A$ 或 $B$ 都从至少 $n-2f$ 个好节点处接收到它所提议的值 ($x$ 或 $y$)，且它们的消息来源没有交集。于是，网络中将至少存在 $2*(n-2f)+f=2n-3f$ 个节点。由于 $3f<n$, 我们将得到 $2n-3f>2n-n=n$，这与前提 (网络中有 $n$ 个节点) 矛盾。

**引理** 4.17. 至少存在一个阶段，该阶段的国王是好节点。

证明. 算法包含 $f+1$ 个阶段，每个阶段的国王都不同。由于系统中只有 $f$ 个拜占庭节点，因此至少一个国王是好节点。

**引理** 4.18. 当 $n>3f$, 如果某一轮的国王是好节点，所有的好节点们在这轮之后都不会改变它们的值 $v$。

证明. 如果所有的好节点都将它们的值修改为国王的值，则所有好节点的值都相同。如果某个好节点没有将它的值改为国王的值，说明它已经至少接收到某个值的提案 $n-f$ 次，于是至少 $n-2f$ 个好节点广播了这个提案。这样，所有的好节点都会接收到这个提议 $n-2f>f$ 次 (因为 $n>3f$)，所以所有好节点都会将它的值设置为这个提案的值，包括国王。注意到根据引理 4.16，只有一个值可以被提议超过 $f$ 次。根据引理 4.15，没有哪个节点会在这一轮之后改变它的值。

**定理** 4.19. 算法 4.14 解决了拜占庭协定问题。

证明. 若以下两个条件之一被满足，国王算法将达成协定。条件一：所有好节点起始的值相同。条件二：在某个阶段 (该阶段的国王为好节点) 之后，所有好节点在同一个值上达成一致 (根据引理 4.17 和引理 4.18)。我们知道好节点将持续保持这个值 (引理 4.15)。算法将确保在 $3*(f+1)$ 轮之后终止。而根据引理 4.15，全部相同有效性肯定成立。

**评论:**

- 算法 4.14 需要 $f+1$ 个预先指派好的国王。我们在之前假设：这些国王及其轮值 king 的顺序，都是给定的。事实上，指派这些国王本身就是一个拜占庭协定任务。而且这个任务必须在国王算法执行之前完成。
- 有没有算法可以在没有国王的情况下实现拜占庭协定？有，请见 4.5 节。
- 我们能否在少于 $f+1$ 轮内解决拜占庭协定 (或至少是共识)？

## 4.4 “轮”数的下界

**定理** 4.20. 在网络中存在 $f$ 个崩溃节点的情况下，一个同步算法需要至少 $f+1$ 轮来达成共识以判定网络中所有节点的最小输入值。

证明. 反证法。假定存在某个算法 $A$，$A$ 可以在 $f$ 轮以内解决共识问题。某个节点 $u_1$ 拥有最小的输入值 $x$, 但是在第一轮中，$u_1$ 在自己崩溃之前，只来得及将自己的信息 (包含它的输入值 $x$) 仅仅发给了另外一个节点 $u_2$。不幸的是，在第二轮，唯一看到 $x$ 的节点 $u_2$ 也仅将 $x$ 发送给了另一个节点 $u_3$，然后崩溃。这个过程不断重复，于是在第 $f$ 轮，只有节点 $u_{f+1}$ 知道最小的值 $x$。由于算法 $A$ 在第 $f$ 轮终止，节点 $u_{f+1}$ 将认为最小值是 $x$，而所有其他存活的 (好) 节点则会接受某个比 $x$ 大的值为最小值。于是算法 $A$ 没有在第 $f$ 轮达成共识。

**评论:**

- 我们也可以得到一个更一般的证明，不需要“判定最小值”这个限制。

- 某些拜占庭节点也可能仅仅崩溃，因此这个下界也对拜占庭协定问题成立。因此算法 4.14 的运行效率是接近最优的。
- 到现在为止，我们所谈到的拜占庭协定算法都假定同步模式。那么，拜占庭协定问题能在异步模式下解决吗？

## 4.5 异步模式下的拜占庭协定算法

**算法 4.21** 异步模式下的拜占庭协定 (Ben-Or, 要求 $f < n/10$)

1: $x_i \in \{0,1\}$ ◁ 本节点的输入值
2: r = 1 ◁ 轮次
3: decided = false
4: 广播 propose($x_i$,r)
5: **repeat**
6: 持续等待，直到接收到 $n-f$ 条当前轮 $r$ 的 propose 消息
7: **if** 至少 $n/2+3f+1$ propose 消息包含同样的值 $x$ **then**
8: $x_i = x$, decided = true
9: **else if** 至少 $n/2+f+1$ propose 消息包含同样的值 $x$ **then**
10: $x_i = x$
11: **else**
12: 随机选择 $x_i$ , 服从 $Pr[x_i = 0] = Pr[x_i = 1] = 1/2$
13: **end if**
14: r = r + 1
15: 广播 propose($x_i$,r)
16: **until** decided (第 8 行)
17: decision = $x_i$

**引理** 4.22. 假定 $n > 9f$，如果一个好节点在算法第 10 行选择了 $x$，那么所有其他的好节点也都会在第 10 行选择 $x$。

证明. 假定 0 和 1 在第 10 行都被选中了。这说明 0 和 1 都被至少 $n/2+1$ 个好节点提议了。因此，网络中将存在至少 $2\cdot n/2+2=n+2>n-f$ 个好节点。与前面矛盾！

**定理** 4.23. 在网络中存在不超过 $f<n/10$ 个拜占庭节点的情况下，算法 4.21 可以解决布尔型拜占庭协定问题 (定义 4.2)。

证明. 首先，由于最多只有 $f$ 个拜占庭节点，在算法第 6 行等待 $n-f$ 条提案消息是没有问题的。

下面证明有效性。如果所有好节点都有相同的输入值 $x$，则所有节点 (除去 $f$ 个拜占庭节点外) 都会提议相同的值 $x$(第 4 行)。于是每个节点都将接收到至少 $n-2f$ 条包含 $x$ 的提案消息。由于 $f<n/10$，可以得到 $n-2f>n/2+3f$。这样在第一轮就将达成一致，将 $x$ 作为决策值。由此可见，我们已经满足了全部相同有效性。若好节点的 (布尔) 输入值不相同，任何决策值都可以符合有效性的要求[4]。

那么一致性能满足吗？假设 $u$ 是第一个选定 $x$ 为决策值的节点 (第 8 行)，且 $u$ 从节点集 $S_u$ (大小为 $n-f$) 处接收到消息 (第 6 行)。由于在异步模式，另一个节点 $v$ 可能从节点集合 $S_v$ (大小为 $n-f$) 处接收到消息。然而，两个集合 $S_u$ 和 $S_v$ 最多只会有 $f$ 个不同的节点[5]。考虑到拜占庭节点可能会撒谎 (向不同的节点发送不同的提案消息)，$v$ 还可能收到额外的 $f$ 条与 $u$ 不同的提案消息。由于节点 $u$ 至少收到了 $n/2+3f+1$ 条包含 $x$ 的提案消息 (第 7 行)，节点 $v$ 将得到至少 $n/2+3f+1-2f=n/2+f+1$ 条包含 $x$ 的提案消息。于是 $v$(即：任意一个好节点) 将在下一轮提议 $x$。

[4]译者注：当好节点的输入值有 0 也有 1 时，决策值为 0 或 1 都可以满足正确输入有效性。

[5]译者注：假设 $S_u$ 和 $S_v$ 有 $f+1$ 个不同的节点。则两个集合一共包含了 $2(n-f)-(f+1)$ 个不同的节点。由于 $n>10f$，我们有 $2(n-f)-(f+1)>n+7f-1$。另一方面，$2(n-f)-(f+1)<n$。这样就有 $n+7f-1<n$，在 $f>0$ 的情况下，这显然是矛盾的。

现在我们只需要关心可终止性了。我们之前已经看到，只要有一个好节点终止运行 (第 8 行)，所有好节点都将在下一轮终止运行。那么，出现某个节点 $u$ 在第 8 行终止的概率有多大呢？首先，根据引理 4.22，一旦有一个好节点进入第 10 行，其他好节点也都会进入第 10 行。那么现在我们主要考虑最坏情况，即：所有好节点都进入第 12 行，随机选择 $x_i$ 的值。显然，我们希望好节点都碰巧选择了同一个值。根据概率公式，所有 $n-f$ 个好节点都选择同一个值的概率为 $2^{-(n-f)+1}$。当这样的情况发生时，所有的好节点都会发送同样的提案消息。因此在最坏情况下，期望的运行时间将随节点数量 $n$ 呈指数增长。

**评论:**

- 这个算法证明了异步拜占庭协定是可以达成的。但不幸的是，由于运行效率过低，该算法没有实际价值。
- 在很长时间内，没有一个算法可以具备亚-指数 (Subexponential) 的时间复杂度。当前最快的算法的时间复杂度为 $O(n^{2.5})$，但是仅仅能容忍 $f \leqslant 1/500n$ 个拜占庭节点。这个算法采用了类似共享硬币算法的思路。此外，节点们还需要尝试探测哪些节点是拜占庭节点。

# 章节说明

最早开始研究拜占庭错误的项目叫 SIFT，该项目由 NASA 资助 [WLG+78]。在 20 世纪 80 年代初，随着 Pease, Shostak 及 Lamport 等人研究成果的发表 [PSL80, LSP82]，拜占庭协定问题开始受到越来越多的关注。在 [PSL80] 中，研究者们提出了算法 4.9 的一般形式，并证明了拜占庭协定问题在 $n \leqslant 3f$ 是无解的。在那篇论文里提出的算法现

在被称为 EIG (*Exponential Information Gathering*)，因为消息的数量是和节点数量呈指数关系。

有很多解决拜占庭协定问题的算法。比如，Queen 算法 [BG89]，它的时间效率要优于 King 算法 [BGP89]，但是只能容忍更少的错误。在 [DS83] 中，提出了一个算法，需要至少 $f+1$ 轮，其理论基础来源于 [FL82]。

虽然有很多同步算法出现，但异步模式要困难得多。现有的异步算法思想都来自于 Ben-Or 和 Brach。Ben-Or 在 [Ben83] 中提出了一个算法，可以容忍 $f < n/5$。Bracha 则对其进行了改进 [BT85]，使得可以容忍 $f < n/3$。在最近几年，才由 King 和 Saia 提出了第一个具备多项式运行时间 (轮数) 的算法 [KS13]，但是这个算法在每个节点的计算时间为指数复杂度。后续的论文 [KS14] 改进了本地的运行效率，从指数时间到多项式时间，代价是只能容忍更少的拜占庭节点 $f < 0.000028n$。

几乎所有算法都只满足全部相同有效性。但也存在一些工作满足其他类型的有效性。比如 [FG03] 的算法在输入值均来自有限域时可以满足正确输入有效性，[SW15] 则可以在输入值为有序时满足中值有效性。

在“拜占庭”这个词出现之前，某些文献曾使用“巴尔巴尼亚将军”或“中国将军”来描述恶意行为。然而最后研究者们都统一采用了拜占庭 [LSP82] 一词。

# 参考文献

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Prin-*

*ciples of distributed computing*, pages 27–30. ACM, 1983.

[BG89] Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415, 1989.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FG03] Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220. ACM, 2003.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. 14(4):183–186, June 1982.

[KS13] Valerie King and Jared Saia. Byzantine agreement in polynomial expected time:[extended abstract]. In *Proceedings of the*

*forty-fifth annual ACM symposium on Theory of computing*, pages 401–410. ACM, 2013.

[KS14] Valerie King and Jared Saia. Faster agreement via a spectral method for detecting malicious behavior. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 785–800, 2014.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[SW15] David Stolz and Roger Wattenhofer. Byzantine Agreement with Median Validity. In *19th International Conference on Priniciples of Distributed Systems (OPODIS), Rennes, France*, 2015.

[WLG+78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. In *Proceedings of the IEEE*, pages 1240–1255, 1978.

# 第 5 章　认证的协定

拜占庭节点能就它们的输入和接收到的消息撒谎。能检测出某些谎言并具有限制拜占庭节点的能力吗？也许可以使用签名来证实消息的真实性。

## 5.1　利用认证的协定

**定义** 5.1 (**签名** (Signature)). 如果一个节点不对消息签名，那么好节点就不会接收该消息。我们用 $\mathtt{msg}(x)_u$ 表示节点 $u$ 签名的消息 $\mathtt{msg}(x)$。

**评论:**

- 算法 5.3给出了在输入值为布尔型时依赖于签名的协定协议。假定存在一个指定的“主”节点 $p$。算法的目标是判定 $p$ 的值。

**定理** 5.2. 算法 5.3 可以容忍 $f < n$ 拜占庭错误，同时在 $f + 1$ 轮终止。

证明. 假定主节点 $p$ 不是拜占庭节点，并且其输入为 1，那么 $p$ 在第一轮广播 $\mathtt{value}(1)_p$，这将触发所有好节点将决策值设为 1。如果 $p$ 的输入是 0，那么不存在签名的消息 $\mathtt{value}(1)_p$，因此无节点可以将决策值设为 1。

**算法 5.3** 使用认证的拜占庭协定

*在主节点 $p$ 上运行的代码:*

1: **if** 输入值为 1 **then**
2: 广播 $\texttt{value}(1)_p$
3: 将 1 设为决策值并终止运行
4: **else**
5: 将 0 设为决策值并终止运行
6: **end if**

*在其他所有节点上 $v$ 上运行的代码:*

7: **for all** rounds $i \in 1, \ldots, f+1$ **do**
8: $S$ 是所有 $\texttt{value}(1)_u$ 的消息集合.
9: **if** $|S| \geqslant i$ 并且 $\texttt{value}(1)_p \in S$ **then**
10: 广播 $S \cup \{\texttt{value}(1)_v\}$
11: 将 1 设为决策值并终止运行
12: **end if**
13: **end for**
14: 将 0 设为决策值并终止运行

如果主节点 $p$ 是拜占庭节点，那么需要所有好节点判定为相同的决策值，否则算法就不正确。假定节点 $p$ 在第 $i$ 轮 $(i < f+1)$ 说服一个好节点 $v$ 其值为 1。由此可知 $v$ 接收到 $i$ 个签名的消息 $\texttt{value}(1)_u$。然后，$v$ 将广播 $i+1$ 个签名的消息 $\texttt{value}(1)_u$ (算法第 10 行)，这将触发所有好节点也判定决策值为 1。如果 $p$ 是在第 $f+1$ 轮才说服某个节点 $v$，则 $v$ 必须接收 $f+1$ 个签名的消息。由于至多 $f$ 个节点是拜占庭节点，至少存在一个好节点 $u$ 在第 $i < f+1$ 轮中已判断决策值为 1，

并发布了消息 $\texttt{value}(1)_u$。这就又回到了前一情形[1]。

**评论:**

- 该算法仅花费 $f+1$ 轮，根据定理 4.20，效率是最优的。
- 利用签名，算法 5.3 可以在 $f < n$ 的情况下达成协定。这是否与定理 4.12矛盾？回忆在定理 4.12 的证明中，我们假定拜占庭节点能就其自己的输入散布彼此矛盾的信息。如果对消息进行了签名，那么好节点能检测出这种行为——对两个互相矛盾的消息进行签名的节点 $u$ 必然是拜占庭节点。
- 算法 5.3 是否满足第 4.1 节介绍的任意一个有效性条件？不满足！若主节点为拜占庭节点，它将能决定决策值。能否对该算法进行修改来使得满足正确输入有效性条件？能！可以对 $2f+1$ 个主节点并行运行该算法。0 或 1 将出现至少 $f+1$ 次。在该情形中，仅能处理 $f < \frac{n}{2}$ 个拜占庭节点。
- 实际上，主节点通常是好节点。如果这样，那么算法 5.3 仅需两轮！能否使该算法适用于任意输入？另外，依赖于同步性限制了该协议的实用性。如果消息可能丢失或者系统是异步的又如何？
- 接下来将介绍 Zyzzyva，它将使用认证的消息来实现状态复制 (定义 2.8)。Zyzzyva 的设计思想是：在好节点上快速运行，在发生错误时将慢下来修复错误[2]！

[1]译者注: 这说明主节点 $p$ 只可能在第 $i < f+1$ 轮说服某好节点其输入值为 1。

[2]译者注：这也是目前主流拜占庭共识算法的一个趋势。

## 5.2 Zyzzyva

**定义** 5.4 (**视图** (View)). 视图 $V$ 描述了一个多副本系统的当前状态，枚举了 $3f+1$ 个副本。视图 $V$ 还将副本之一标记为主节点 $p$。

**定义** 5.5 (**命令** (Command)). 如果客户端要更新 (或者读取) 数据，该客户端在一条 `Request` 消息中发送一个适当的命令 $c$ 到主节点 $p$。除命令 $c$ 自身之外，该 `Request` 消息还包括一个时间戳[3] $t$。客户端对该消息签名来保证其真实性 (Authenticity)。

**定义** 5.6 (**历史** (History)). 历史 $h$ 是一个命令序列 $c_1, c_2, \ldots$，这些命令按照它们被 Zyzzyva 执行的顺序排列。若一个历史的最后一个命令为 $c_k$，则该历史被记为 $h_k$。

**评论:**

- 在 Zyzzyva 中，主节点 $p$ 被用来对客户端提交的命令进行排序，以创建一个历史 $h$。
- 除全局接受的历史之外，节点 $u$ 还可以具有本地历史，表示为 $h^u$ 或者 $h_k^u$。

**定义** 5.7 (**已完成的命令** (Complete Command)). 如果一条命令完成了，那么它将保持在历史 $h$ 中它自己的位置处，即使发生错误也是如此。

**评论:**

- 只要客户端在等待他们的命令完成，就可能将 Zyzzyva 作为一个单个计算机对待，即使存在至多 $f$ 个错误节点也是如此。

[3]译者注：时间戳是不可缺少的，否则副本们就不知道一条命令到底是需要被执行两次，还是说一条重复发送的命令。

## 无错误的情况

**算法 5.8** Zyzzyva: 无错误情况

1: 在时刻 $t$，客户端 $u$ 希望执行命令 $c$
2: 客户 $u$ 将请求 $\mathtt{R} = \mathtt{Request}(c, t)_u$ 发送到主节点 $p$
3: 主节点 $p$ 将 $c$ 增加到它本地历史中, $h^p = (h^p, c)$
4: 主节点 $p$ 向所有副本发送 $\mathtt{OR} = \mathtt{OrderedRequest}(h^p, c, \mathtt{R})_p$
5: 每个副本 $r$ 将命令 $c$ 增加到它本地历史中，$h^r = (h^r, c)$，并且检查 $h^r = h^p$ 是否成立
6: 每个副本 $r$ 执行命令 $c_k$，并得到运行结果 $a$
7: 每个副本 $r$ 向客户端 $u$ 发送 $\mathtt{Response}(a, \mathtt{OR})_r$
8: 客户端 $u$ 将接收到的 $\mathtt{Response}(a, \mathtt{OR})_r$ 消息存放到集合 $S$ 中
9: 客户端 $u$ 检查所有的历史 $h^r$ 是否一致
10: **if** $|S| = 3f + 1$ **then**
11: 　客户端 $u$ 认为命令 $c$ 已完成
12: **end if**

**评论:**

- 由于客户端接收到 $3f + 1$ 个一致的响应，所以所有正确的副本都必然处于同一状态。
- 完成命令 $c$ 仅需要三个通信回合。
- 注意，副本并不清楚哪些命令被客户端认为已完成！如何确保客户端认为已完成的命令实际被执行了？我们将在引理 5.23 中讨论。
- 从客户端接收的命令应当根据时间戳排序来保持命令的因果顺序。

- 有许多有待优化的地方。例如，在大多数消息中包含完整的命令历史引入了很大的开销。相反，可以删除一致同意的历史。另外，发送历史的其余部分的哈希值已足够检查副本之间的一致性了。

- 如果客户端未接收到 $3f+1$ 个 $\texttt{Response}(a, \texttt{OR})_r$ 消息，会出现什么情况？一个拜占庭副本可能完全不发送任何内容。实践中，客户端可设置一个计时器 (Timeout) 用于收集 Response 消息。这是否意味着 Zyzzyva 仅工作在同步模式中？是，也不是。我们将在引理 5.26 和引理 5.27 中对此进行讨论。

## 拜占庭副本

**算法 5.9** Zyzzyva: 拜占庭副本 (接算法 5.8)

1: **if** $2f+1 \leqslant |S| < 3f+1$ **then**
2: 客户端 $u$ 向所有副本发送 $\texttt{Commit}(S)_u$
3: 每个副本 $r$ 向 $u$ 回复 $\texttt{LocalCommit}(S)_r$
4: 客户端 $u$ 收集至少 $2f+1$ 条 $\texttt{LocalCommit}(S)_r$ 消息，于是认为命令 $c$ 已经完成
5: **end if**

**评论:**

- 如果有副本宕机，一个客户端 $u$ 将接收到少于 $3f+1$ 条一致的相应消息。客户端 $u$ 只能假设如果所有正确副本最终将命令 $c$ 附加到它们的本地历史 $h^r$，那么客户端 $u$ 只能假设命令 $c$ 是完成的。

**定义** 5.10 (**提交证书** (Commit Certificate)). 一个提交证书 (Commit Certificate) $S$ 包含来自 $2f+1$ 个不同副本 $r$ 的 $2f+1$ 个一致的且已签名的消息 $\texttt{Response}(a, \texttt{OR})_r$。

**评论:**

- 集合 $S$ 是一个提交证书，该提交证书证明命令在 $2f+1$ 个副本上执行并且其中的 $f+1$ 个正确。在客户端认为命令完成前，该提交证书 $S$ 必须被 $2f+1$ 个副本确认。
- 为何客户端必须向 $2f+1$ 个副本分发该提交证书？将在引理 5.21 中讨论。
- 如果 $|S| < 2f+1$ 怎么办？或者如果客户端接收到 $2f+1$ 条消息但是其中一些具有不一致的历史又该怎么办？由于至多 $f$ 个副本是拜占庭节点，所以此时主节点自身必然是拜占庭节点！能解决这个问题吗？

## 拜占庭主节点

**定义** 5.11 (**不当行为证明** (Proof of Misbehavior)). 一组互相矛盾的签名的消息可以构成对一些节点的不当行为的证明。

**评论:**

- 例如，如果客户端 $u$ 接收到两条由主节点签名的消息 $\texttt{Response}(a, \texttt{OR})_r$，但其中包含的 OR 并不一致。那么客户端 $u$ 能够证明主节点行为不当。客户端 $u$ 通过向所有副本广播一条 $\texttt{IHatePrimary}_r$ 消息来通告这个不当行为，同时也启动了一个视图改变 (View Change)。

**算法 5.12** Zyzzyva: 拜占庭主节点 (接算法 5.9)

1: **if** $|S| < 2f + 1$ **then**
2: 客户端 $u$ 向所有副本发送原始的 $\mathtt{R} = \mathtt{Request}(c, t)_u$ 消息
3: 每个副本 $r$ 向主节点 $p$ 发送一条 $\mathtt{ConfirmRequest}(\mathtt{R})_r$
4: **if** 主节点 $p$ 回复消息 OR **then**
5: 副本 $r$ 将 OR 转发给所有其他副本
6: 从第 5 行开始继续运行算法 5.8
7: **else**
8: 副本 $r$ 广播 $\mathtt{IHatePrimary}_r$ 以启动视图改变 (View Change)
9: **end if**
10: **end if**

**评论:**

- 拜占庭主节点 (Faulty Primary) 可以通过不发出 OrderedRequest 消息 (算法 5.8 第 4 行) 来降低 Zyzzyva 的速度，并重复进入到算法 5.12。[4]
- 该算法中的第 5 行对于确保存活 (Liveness) 是必须的。将在定理 5.27中对此进行讨论。
- 仍然存在优化的空间。例如，一个副本可能已知道客户端请求的命令。在该情形中，它可以作出回答而不询问主节点。进一步地，主节点可能已知道副本请求的消息 OR。此时，它将旧的 OR 消息发送给发出请求的复本。

[4]译者注：若主节点不发送，或没有向所有副本发送 OrderedRequest，客户端接收到的 $\mathtt{Response}(a, \mathtt{OR})_r$ 消息就会减少。

## 安全性

**定义** 5.13 (**安全性** (Safety)). 如果下面的条件成立，则称系统是安全的：如果具有序列号 $j$ 和历史 $h_j$ 的命令完成，那么对于较早完成的任意命令 (具有较小的序列号 $i < j$)，历史 $h_i$ 是历史 $h_j$ 的前缀 (Prefix)。

**评论:**

- 在 Zyzzyva 中，一条命令仅可以以两种方式完成，或者在算法 5.8中或者在算法 5.9中。
- 如果系统是安全的，那么已完成的命令不能被重排序或丢弃。那么到目前为止我们讨论的 Zyzzyva 安全么？

**引理** 5.14. 若 $c_i$ 和 $c_j$ 是两条不同的已完成的命令。则 $c_i$ 和 $c_j$ 的序列号必然不同。

证明. 如果命令 $c$ 在算法 5.8 中完成，那么 $3f+1$ 个副本向客户端发送 $\texttt{Response}(a, \texttt{OR})_r$ 消息。若命令 $c$ 在算法 5.9 中完成，至少 $2f+1$ 个副本向客户端发送 $\texttt{Response}(a, \texttt{OR})_r$ 消息。因此，客户端必然接收到至少 $2f+1$ 条 $\texttt{Response}(a, \texttt{OR})_r$ 消息。

命令 $c_i$ 和 $c_j$ 都是完成的。因此必然有至少 $2f+1$ 个副本向 $c_i$ 发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息。但是还有至少 $2f+1$ 个副本向 $c_j$ 发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息。因为只有 $3f+1$ 个副本，至少一个好的副本 (非拜占庭) 向 $c_i$ 和 $c_j$ 都发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息。好的副本对于每个序列号仅发送一条 $\texttt{Response}(a, \texttt{OR})_r$ 消息，因此这两条命令必须具有不同的序号。

**引理** 5.15. 令 $c_i$ 和 $c_j$ 为两条已完成的命令，且序号 $i < j$。则历史 $h_i$ 是历史 $h_j$ 的前缀。

证明. 正如在对引理 5.14 的证明中所述，必然存在至少一个好的副本对 $c_i$ 和 $c_j$ 都发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息。

若一个好的副本 $r$ 对 $c_i$ 发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息，则 $r$ 接受 $c_j$ 必须满足如下条件：主节点所提供的 $c_j$ 的历史与副本 $r$ 的本地历史 (本地历史已包含了 $c_i$) 相一致。

**评论:**

- 拜占庭主节点可以采取如下手段来导致系统不完成任何命令: 不发送任何消息或者对客户端请求进行不一致的排序。在该情形中，必须选择其他副本来替代主节点。

## 视图改变

**定义** 5.16 (**视图改变** (View Change)). 在 Zyzzyva 中，视图改变的目的是用另一个 (有希望是正确的) 副本来替代拜占庭主节点。视图改变由某些副本发起，它们向所有其他副本发送 $\texttt{IHatePrimary}_r$ 消息以启动一个视图改变。视图改变可以由下面两个条件之一触发：(1) 副本从一个客户端得知，主节点存在不当行为 (定义 5.11); (2) 副本不能收到主节点发出的 OR 消息 (算法 5.12)。

**评论:**

- 如何才能安全地决定发起视图改变，将拜占庭主节点降级为普通的副本？须注意，不能让拜占庭节点触发视图改变!

**算法 5.17** Zyzzyva: 视图改变协定

1: 所有副本持续接收 $\texttt{IHatePrimary}_r$ 消息，并将它们存放于集合 $H$ 中
2: **if** 某副本 $r$ 接收到了 $|H| > f$ 消息或者接收到一条有效的 ViewChange 消息 **then**
3: 副本 $r$ 广播 $\texttt{ViewChange}(H^r, h^r, S_l^r)_r$
4: 副本 $r$ 停止参与当前的视图
5: 副本 $r$ 切换到下一个主节点 "$p = p + 1$"
6: **end if**

**评论:**

- 集合 $H$ 中的 $f + 1$ 个 $\texttt{IHatePrimary}_r$ 消息证明至少一个好副本发起了视图改变。该证明被广播到所有副本来确保一旦第一个好副本在当前视图中停止活动，所有其他副本都将跟随。
- $S_l^r$ 是副本在即将结束的视图中所获得的最近的提交证书 (算法 5.9)。$S_l^r$ 将被用来在新的视图开始前恢复正确历史。若一个正确的客户端从副本接收到 $3f + 1$ 个响应，则对应的命令应被标记为已完成。本地历史 $h_r$ 被包括在 $\texttt{ViewChange}(H^r, h^r, S_l^r)_r$ 消息中，使得这些已完成的命令可以被恢复。
- 在 Zyzzyva 中，拜占庭主节点在视图改变后开始充当常规副本。实践中，所有机器最终都损坏并且很少在此之后修复自身。作为替代，可以考虑采用不在前一视图中的新副本来替换拜占庭主节点。

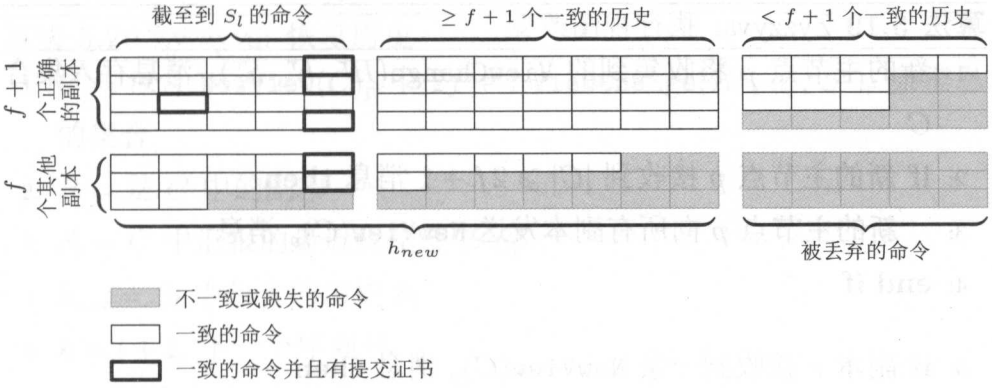



图 5.18: 图中展现的是 $C$ 中不同副本汇报的数据的结构。直到上一个提交证书 $S_1$ 之前的命令已在算法 5.8或算法 5.9中完成。在上一个提交证书 $S_1$ 之后，可能有命令在一个正常的客户端处完成 (算法 5.8)。而算法 5.20 展示了新的历史 $h_{new}$ 如何恢复，使得所有完成的命令都不会丢失。

**评论:**

- 与引理 5.15 类似，提交证书是有序的。对于具有序号 $i < j$ 的两个提交证书 $S_i$ 和 $S_j$，由 $S_i$ 鉴证的历史 $h_i$ 是由 $S_j$ 鉴证的历史 $h_j$ 的前缀。
- Zyzzyva 收集最近提交证书和 $2f+1$ 个副本的本地历史。该信息被分发给所有副本，并且用来恢复新视图的历史 $h_{new}$。
- 如果副本未及时接收到 $\texttt{NewView}(C)_p$ 或 $\texttt{ViewConfirm}(h_{\texttt{new}})_r$ 消息，那么它将向所有其他副本广播 $\texttt{IHatePrimary}_r$ 消息来触发另一次视图改变。
- 完全恢复的历史看起来怎么样？看起来 $C$ 中包括的历史的集合可能是凌乱的。如何能确信已完成的命令未被重排序或者丢弃？

**算法 5.19** Zyzzyva: 执行视图改变

1: 新的主节点 $p$ 将收集到的 ViewChange$(H^r, h^r, S_l^r)_r$ 消息存入集合 $C$
2: **if** 新的主节点 $p$ 接收到 $|C| \geqslant 2f+1$ 消息 **then**
3: 新的主节点 $p$ 向所有副本发送 NewView$(C)_p$ 消息
4: **end if**

5: **if** 副本 $r$ 接收到一条 NewView$(C)_p$ 消息 **then**
6: 副本 $r$ 恢复新的历史 $h_{\texttt{new}}$ (将在算法 5.20中介绍)
7: 副本 $r$ 向所有副本广播 ViewConfirm$(h_{\texttt{new}})_r$
8: **end if**

9: **if** 副本 $r$ 接收到 $2f+1$ ViewConfirm$(h_{\texttt{new}})_r$ 消息 **then**
10: 副本 $r$ 采纳 $h^r = h_{\texttt{new}}$ 作为新视图的历史
11: 副本 $r$ 开始参与新的视图
12: **end if**

**评论:**

- 直至 $S_l$ 之前的命令被包括到新的历史 $h_{new}$ 中。
- 如果最后提交证书 $S_l$ 之后至少 $f+1$ 个副本共享一致的历史，那么之后的命令也将被包括进去。
- 即使 $f+1$ 个正确副本一致地报告最后的提交证书 $S_l$ 之后的命令 $c$，客户端也可能不认为 $c$ 是已完成的，因为发到该客户端的某条响应被丢失了。这样的命令被包括到新的历史 $h_{new}$ 中。当客户端重新尝试执行 $c$ 时，副本将能够利用该客户端的请求中包括的时间戳标识出同一命令 $c$，从而避免重复执行该命令。
- 能确保在正确的客户端处完成的所有命令都被延续到新视图

**算法 5.20** Zyzzyva: 恢复历史

1: $C$ = 包含 $\texttt{NewView}(C)_p$ 中 $2f+1$ $\texttt{ViewChange}(H^r, h^r, S^r)_r$ 条消息的集合
2: $R$ = 是 $C$ 中的副本
3: $S_l = C$ 中汇报的最近提交证书 $S_l^r$
4: $h_{\texttt{new}} = S_l$ 中包含的历史 $h_l$
5: $k = l+1$, 下一个序列号
6: **while** 命令 $c_k$ 存在于 $C$ **do**
7: **if** $c_k$ 被 $R$ 中至少 $f+1$ 个副本汇报 **then**
8: 从 $R$ 中删除那些不支持 $c_k$ 的副本
9: $h_{\texttt{new}} = (h_{\texttt{new}}, c_k)$
10: **end if**
11: $k = k+1$
12: **end while**
13: return $h_{\texttt{new}}$

吗?

**引理** 5.21. 全局最近提交证书 $S_l$ 被包括在 $C$ 中。

证明. 任意两个包含 $2f+1$ 个副本的集合共享至少一个正确副本。因此，确认了最近提交证书 $S_l$ 的至少一个正确副本也发送了在 $C$ 中的 $\texttt{LocalCommit}(S_l)_r$ 消息。

**引理** 5.22. 在 $S_l$ 之后完成的任意命令及其历史必然在 $C$ 中被报告至少 $f+1$ 次。

证明. 命令 $c$ 可以仅在 $S_l$ 之后在算法 5.8中完成。因此，$3f+1$ 个副本向 $c$ 发送了 $\texttt{Response}(a, \texttt{OR})_r$ 消息。$C$ 包括 $2f+1$ 个副本的本地历史，

在这 $2f+1$ 个副本中至多 $f$ 个副本是拜占庭节点。于是，$c$ 及其历史可以在 $C$ 中的至少 $f+1$ 个本地历史中发现。

**引理** 5.23. 如果客户端认为命令 $c$ 是完成的，那么在视图改变期间命令 $c$ 在历史中的位置保持不变。

证明. 在引理 5.21 中示出了最近提交证书包含在 $C$ 中，因此在算法 5.9 中终止的任何命令都被包括在视图改变之后的新的历史中。结果，在最后提交证书 $S_l$ 之前完成的每条命令都被包括在该历史中。

在最后提交证书之后，且在算法 5.8 中完成的命令由至少 $f+1$ 个正确副本支持 (引理 5.22)。如算法 5.20 所述，这种命令被添加到该新的历史中。算法 5.20 顺序添加命令，直到历史变得不一致。因此，在视图改变期间完成的命令未被丢失或者重排序。

**定理** 5.24. 即使在视图改变期间 Zyzzyva 也是安全的。

证明. 如引理 5.15中所述，在一个视图内已完成的命令未被重新排序。另外，如在引理 5.23中所示，在视图改变期间完成的命令也不会被丢失或者重排序。因此，Zyzzyva 是安全的。

**评论:**

- 因此即使在存在错误时 Zyzzyva 也正确地处理了已完成的命令。我们还希望 Zyzzyva 更进一步，即，正确客户端所发布的命令应当最终完成。
- 如果网络断开或者引入任意大的延迟，那么命令可能永远不会完成。
- 若延迟时间是有限的，能确保命令完成吗？

**定义** 5.25 (**活性** (Liveness)). 如果每一条命令最终都完成，那么称系统是活的 (*Live*)。

**引理** 5.26. 如果主节点是正确的并且命令是由正确的客户端请求的，则 Zyzzyva 在同步期间是活的，

证明. 客户端从所有正确副本接收 $\texttt{Response}(a, \texttt{OR})_r$ 消息。如果它接收到 $3f+1$ 条消息，命令在算法 5.8中立即完成。如果客户端接收到少于 $3f+1$ 条消息，那么它将至少接收 $2f+1$ 条消息，这是因为存在至多 $f$ 个拜占庭副本。所有正确副本都将用正确的 $\texttt{LocalCommit}(S)_r$ 消息回答客户端的 $\texttt{Commit}(S)_u$ 消息，此后该命令在算法 5.9 中完成。

**引理** 5.27. 如果在同步时段期间，一个请求未在算法 5.8 或算法 5.9 中完成，那么将发生视图改变。

证明. 如果一条命令在足够长的时间内未完成，客户端将向所有副本重新发送 $\texttt{R} = \texttt{Request}(c, t)_u$ 消息。此后，如果一个副本的 $\texttt{ConfirmRequest}(\texttt{R})_r$ 消息未能得到主节点的及时答复，那么它就广播 $\texttt{IHatePrimary}_r$ 消息。

如果正确副本收集到 $f+1$ 条 $\texttt{IHatePrimary}_r$ 消息，那么视图改变就被发起。

如果没有正确副本收集到多于 $f$ 条 $\texttt{IHatePrimary}_r$ 消息，那么至少一个正确副本接收到从主节点转发给所有其他副本的有效 $\texttt{OrderedRequest}(h^p, c, \texttt{R})_p$ 消息。在该情形中，客户端可确信接收到来自正确副本的至少 $2f+1$ 条 $\texttt{Response}(a, \texttt{OR})_r$ 消息，并且通过组装提交证书来完成该命令。

**评论:**

- 如果新选出的主节点是拜占庭节点，那么视图改变可能不会终止。然而，当所有包含的消息都有签名时，我们可判断新的主节点是否正确地组装了 $C$。如果主节点拒绝组装 $C$，那么副本在超时之后将发起另一次视图改变。

## 章节说明

Dolev 等 [DFF+82] 在 1982 年介绍了算法 5.3。

有很多协议来解决拜占庭容错状态机复制 (BFT) 问题。

Castor 和 Liskov [MC99] 在 1999 年介绍了实际拜占庭容错 (PBFT) 协议[5]，后续工作包括 Farsite [ABC+02]。在也带动了其他工作的发展，比如 Q/U [AEMGG+05]，以及 HQ [CML+06]。

Zyzzyva [KAD+07] 对性能的提高主要体现在没有错误的情形，而 Aardvark [CWA+09] 提高了有错误情况下的性能。Guerraoui 等 [GKQV10] 介绍了模块化系统，基于该系统可以更容易开发出鲁棒性或者最佳情况性能方面匹配具体应用的 BFT 协议。

## 参考文献

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage

[5]译者注：PBFT 是广泛用于联盟链的共识算法。

for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CWA+09] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[DFF+82] Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[GKQV10] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[MC99] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

# 第 6 章　仲裁系统

如果单个服务器的能力不再足以服务所有客户，会发生什么情况呢？直观的选项是增加更多的服务器，并且使用过半数方法 (Majority Approach) (例如，第 2章的 Paxos) 来保证一致性。然而，即使购买一百万台服务器，客户端仍旧不得不在每次请求中访问超过一半的服务器！尽管收获了容错能力，但是性能至多被翻倍。我们是否必须得放弃一致性？

让我们回退一步：我们使用过半数方法是因为两个过半数集合总是重叠。然而过半数集合是保证重叠的仅有方法吗？本章中，我们研究重叠集合背后的理论，称作仲裁系统 (Quorum System)。

**定义** 6.1 (**仲裁** (Quorum), **仲裁系统** (Quorum System)). 令 $V = \{v_1, \ldots, v_n\}$ 为节点集合。一个仲裁 (Quorum) $Q \subseteq V$ 是所有节点的一个子集。一个仲裁系统 (Quorum System) $\mathcal{S} \subset 2^V$ 是由多个仲裁构成的集合，且两两交集不为空，即, $Q_1 \cap Q_2 \neq \emptyset \ \forall Q_1, Q_2 \in \mathcal{S}$.

**评论:**

- 当使用仲裁系统时，客户端选择一个仲裁，获取该仲裁内所有节点上的锁 (或者票 ticket)，然后在完成作业之后再释放所有锁。该想法在于不管挑选哪一个仲裁，该仲裁内的节点都将与其他每一个仲裁的节点重叠。
- 如果两个仲裁试图同时对它们的节点加锁，会发生什么情况？

- 若一个仲裁系统 $\mathcal{S}$ 满足如下情况，则称其为极小的 (minimum)：$\forall Q_1, Q_2 \in \mathcal{S} : Q_1 \not\subset Q_2$.
- 最简单的仲裁系统仅包含一个仲裁，且该仲裁只有一个服务器节点。该系统称作单节点仲裁 (Singleton)。
- 在过半数 (Majority) 仲裁系统中，每个仲裁具有 $\lfloor \frac{n}{2} \rfloor + 1$ 个节点。
- 能否想出其他简单的仲裁系统？

## 6.1 负载和工作量

**定义** 6.2 (**访问策略** (Access Strategy)). 给定一个仲裁系统 $\mathcal{S}$，一个访问策略 $Z$ 定义了访问 $\mathcal{S}$ 中每个仲裁 $Q$ 的概率 $P_Z(Q)$，须满足 $\sum_{Q \in \mathcal{S}} P_Z(Q) = 1$。

**定义** 6.3 (**负载** (load)).

- 一个访问策略 $Z$ 在一个节点 $v_i$ 上的负载 (load) 为 $L_Z(v_i) = \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q)$。
- 由一个访问策略 $Z$ 引发的在一个仲裁系统 $\mathcal{S}$ 上的负载被定义为由 $Z$ 引发的在 $\mathcal{S}$ 中任意一个节点上的最大负载，即 $L_Z(\mathcal{S}) = \max_{v_i \in \mathcal{S}} L_Z(v_i)$。
- 一个仲裁系统 $\mathcal{S}$ 的负载被定义为 $L(\mathcal{S}) = \min_Z L_Z(\mathcal{S})$。

**定义** 6.4 (**工作量** (work)).

- 一个仲裁 $Q \in \mathcal{S}$ 的工作量 (work) 是该仲裁中的所有节点的数量，即：$W(Q) = |Q|$。

- 由一个访问策略 $Z$ 引发的在一个仲裁系统 $\mathcal{S}$ 上的工作量是平均 (期望) 被访问的节点个数，$W_Z(\mathcal{S}) = \sum_{Q\in\mathcal{S}} P_Z(Q)\cdot W(Q)$。
- 一个仲裁系统 $\mathcal{S}$ 的工作量为 $W(\mathcal{S}) = \min_Z W_Z(\mathcal{S})$。

**评论:**

- 注意，在计算工作量和负载时，不能采用不同的访问策略 $Z$，必须对这二者使用单个 $Z$。
- 下面用一个小例子讨论上面的概念。设 $V = \{v_1, v_2, v_3, v_4, v_5\}$，$\mathcal{S} = \{Q_1, Q_2, Q_3, Q_4\}$，且 $Q_1 = \{v_1, v_2\}$, $Q_2 = \{v_1, v_3, v_4\}$, $Q_3 = \{v_2, v_3, v_5\}$，$Q_4 = \{v_2, v_4, v_5\}$。如果我们选择访问策略 $Z$，使得 $P_Z(Q_1) = 1/2$ 且 $P_Z(Q_2) = P_Z(Q_3) = P_Z(Q_4) = 1/6$，则具有最大负载的节点为 $v_2$，其负载 $L_Z(v_2) = 1/2 + 1/6 + 1/6 = 5/6$, i.e., $L_Z(\mathcal{S}) = 5/6$。相应地，我们可以计算工作量为 $W_Z(\mathcal{S}) = 1/2\cdot 2 + 1/6\cdot 3 + 1/6\cdot 3 + 1/6\cdot 3 = 15/6$。
- 能否为 $\mathcal{S}$ 设计一个更好的访问策略？
- 如果仲裁系统 $\mathcal{S}$ 中的每个仲裁 $Q$ 具有相同数目的节点，那么称 $\mathcal{S}$ 是均匀的 (Uniform)。
- 一个仲裁系统可以具有的最小负载为多少？

| 主拷贝 vs. 过半数 | 单节点仲裁 (Singleton) | 过半数 (Majority) |
|---|---|---|
| 需要访问的节点数? (工作量) | **1** | $> n/2$ |
| 最繁忙节点的负载? (负载) | 1 | $> \mathbf{1/2}$ |

表 6.5: 对单节点仲裁 (Singleton) 和过半数仲裁系统 (Majority Quorum Systems) 的比较。须注意当每个单一节点出错概率大于 1/2 时，单节点仲裁系统可能是一个好的选择。

**定理** 6.6. 设 $\mathcal{S}$ 是一个仲裁系统，则 $L(\mathcal{S}) \geqslant 1/\sqrt{n}$ 必然成立。

证明. 假设 $Q = \{v_1, \ldots, v_q\}$ 是 $\mathcal{S}$ 中最小的仲裁，且 $|Q| = q$，$|\mathcal{S}| = s$。设 $Z$ 是定义在 $\mathcal{S}$ 上的一个访问策略.

$\mathcal{S}$ 中其他每个仲裁都包含仲裁 $Q$ 中至少一个节点。于是当任意一个仲裁被访问，$Q$ 中至少一个节点也会被访问，这样就可以得出一个下界：至少存在一个节点 $v_i \in Q$，使得[1] $L_Z(v_i) \geqslant 1/q$。

此外，由于 $Q$ 是最小的仲裁，则至少有 $q$ 个节点会被访问，于是[2] $W(\mathcal{S}) \geqslant q$。

由于每次访问 $q$ 个节点，访问最频繁的节点的负载将至少为 $q/n$，这就得到另一个下界[3]：至少存在一个节点 $v_i \in Q$，使得 $L_Z(v_i) \geqslant q/n$。

综合上述结论，我们就得到了[4] $L_Z(\mathcal{S}) \geqslant \max(1/q, q/n) \Rightarrow L_Z(\mathcal{S}) \geqslant 1/\sqrt{n}$。

由于 $Z$ 可以为任意的访问策略，我们就可以得出 $L(\mathcal{S}) \geqslant 1/\sqrt{n}$。

**评论:**

- 有可能实现这样的负载吗?

[1]译者注：若所有节点的负载都小于 $1/q$，则访问 $Q$ 中任意节点的概率就小于 1. 但是无论访问哪个仲裁，$Q$ 中至少都有一个节点被访问。矛盾。

[2]译者注：$W(\mathcal{S}) \geqslant W_Z(\mathcal{S}) = \sum_{Q \in \mathcal{S}} P_Z(Q) \cdot W(Q) \geqslant q \cdot \sum_{Q \in \mathcal{S}} P_Z(Q) = q$。

[3]译者注：注意 $W(\mathcal{S})$ 就是平均访问的节点个数，而负载 $L_Z(v_i)$ 实际是节点 $v_i$ 被访问的概率。若所有节点的负载均小于 $q/n$，我们就可以得到平均访问的节点个数小于 $n \cdot q/n = q$。矛盾。

[4]译者注：此处我们只需要理解：$1/q$ 和 $q/n$ 不可能同时大于 $1/\sqrt{n}$。若 $q < \sqrt{n}$，则 $1/q > 1/\sqrt{n}$；否则，$q \geqslant \sqrt{n}$，$q/n \geqslant \sqrt{n}/n = 1/\sqrt{n}$。

## 6.2 网格仲裁系统

**定义** 6.7 (**基本的网格仲裁系统** (Basic Grid Quorum System)). 假设 $\sqrt{n} \in \mathbb{N}$，将 $n$ 个节点布置到边长为 $\sqrt{n}$ 的方形矩阵中，即网格。基本网格仲裁系统由 $\sqrt{n}$ 个仲裁组成，每个仲裁包含完整的第 $i$ 行和完整的第 $i$ 列，$1 \leqslant i \leqslant \sqrt{n}$。

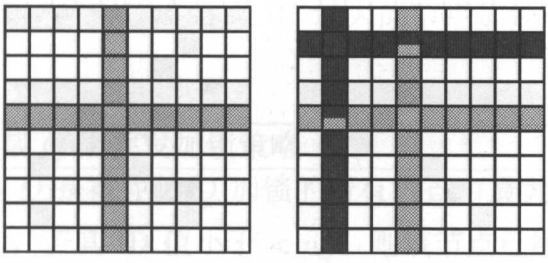


图 6.8: 基本的网格仲裁系统。每个仲裁 $Q_i$ 包含第 $i$ 行和第 $i$ 列的节点 $(1 \leqslant i \leqslant \sqrt{n})$。每个仲裁包含的节点数均为 $2\sqrt{n}-1$，且任意两个仲裁正好有两个重复的节点。于是，当访问策略 $Z$ 是均匀的 (即：访问每个仲裁的概率都为 $1/\sqrt{n}$) 时，该仲裁系统的工作量为 $2\sqrt{n}-1$，且每个节点的负载均为 $\Theta(1/\sqrt{n})$。

**评论:**

- 考虑图 6.8中的右侧图片，两个仲裁有两个重复的节点。如果要同时访问这两个仲裁，则不能保证至少一个仲裁对它的所有节点加锁，因为它们可能进入死锁！
- 如果只有两个仲裁，可以通过使它们仅在一个节点上相交来解决此问题，参见图 6.9。但是，若有三个仲裁，死锁依然可能发生，无法保证程序顺利运行!
- 但是，如果节点是完全排序的，那么通过避免“一次访问所有节点”的策略，我们能够保证程序顺利运行！

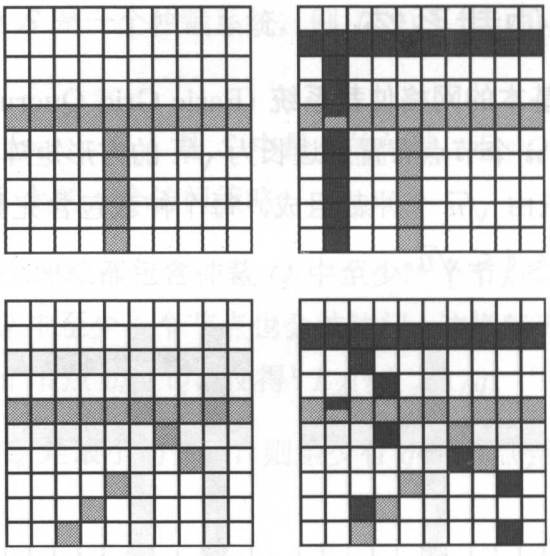


图 6.9: 存在其他方式在网格中挑选仲裁，使得两个不同仲裁仅有一个重复的节点。每个仲裁的大小介于 $\sqrt{n}$ 和 $2\sqrt{n}-1$ 之间，而工作量为 $\Theta(\sqrt{n})$。当访问策略 $Z$ 是均匀的时，每个节点的负载均为 $\Theta(1/\sqrt{n})$。

**算法 6.10** 对仲裁 $Q$ 顺序加锁的策略

1: 尝试逐一对节点加锁，以节点 ID 为序
2: 若某个节点已被锁住了，就释放所有的锁并重新开始

**定理** 6.11. 如果用算法 6.10访问每个仲裁，那么至少一个仲裁将获得它的所有节点的锁。

证明. 我们通过反证来证明该定理。假设没有仲裁能继续运行，即：对于每个仲裁，其节点中的至少一个被另一个仲裁加锁。

令 $v$ 为被某一仲裁 $Q$ 加锁的节点中 ID 值最大的。须注意 $Q$ 已对其节点中 ID 值比 $v$ 小的所有节点加锁，否则 $Q$ 将会重新开始。由于 ID 值比 $v$ 大的所有节点都未被加锁，所以 $Q$ 要么已对其所有节点加锁，要么能继续运行——这就导出了矛盾。由于节点的集合是有限的，

所以存在一个仲裁，它最终能对其所有节点加锁。

**评论:**

- 难道现在我们返回到了分布式系统中的顺序访问？让我们用同样的思想来并发地解决此问题，即，通过节点的排序来解决冲突。那么，已对迄今为止 ID 最大的节点加锁的仲裁总是能继续运行。

---

**算法 6.12** 对仲裁 $Q$ 的并发加锁策略

---

**不变量**: 设 $v_Q \in Q$ 是被仲裁 $Q$ 加锁的所有节点中最大的 ID 值[5]，即：任何节点 $v_i \in Q$，若其 ID 值小于 $< v_Q$，则该节点已经被 $Q$ 加锁。若 $Q$ 还没有得到任何锁，则 $v_Q$ 设为 0。

1: **repeat**
2: 仲裁 $Q$ 尝试对其所有节点加锁
3: **for each** 不能被 $Q$ 加锁的节点 $v \in Q$ **do**
4: 若 $Q'$ 是目前成功对 $v$ 加锁的仲裁，与 $Q'$ 交换 $v_Q$ and $v_{Q'}$
5: **if** $v_Q > v_{Q'}$ **then**
6: $Q'$ 释放对 $v$ 的锁，且 $Q$ 获得对 $v$ 的锁[6]
7: **end if**
8: **end for**
9: **until** 仲裁 $Q$ 中所有节点均被加锁

---

**定理** 6.13. 如果节点和仲裁都使用算法 6.12，那么至少一个仲裁将获得用于其所有节点的锁。

证明. 证明过程类似于定理 6.11的证明：假设没有仲裁能继续运行。然而，至少具有最大 $v_Q$ 的仲裁总是可以继续运行的-矛盾！由于节点集合是有限的，至少一个仲裁将最终能够获得其所有节点上的锁。

**评论:**

- 如果一个仲裁对其所有节点加锁然后崩溃了会怎么样？该仲裁系统死掉了？我们可以解决这个问题，比如：通过使用租约而不是锁。租约具有时限 (timeout)，即，锁最终会被释放。

## 6.3 容错

**定义** 6.14 (**适应力** (Resilience)). 若一个仲裁系统 $\mathcal{S}$，在任意 $f$ 个节点发生故障的情况下，仍存在至少一个无故障节点的仲裁 $Q \in \mathcal{S}$，则称 $\mathcal{S}$ 是 $f$-适应的。满足上述条件的最大的 $f$ 值被称为适应力 (Resilience)，记为 $R(\mathcal{S})$。

**定理** 6.15. 令 $\mathcal{S}$ 为网格仲裁系统，包含 $n$ 个仲裁，每个仲裁均由完整的一行和完整的一列组成。则 $\mathcal{S}$ 的适应力为 $\sqrt{n}-1$。

证明. 如果网格对角线上的所有 $\sqrt{n}$ 个节点都发生故障，那么每个仲裁将具有至少一个故障节点。如果少于 $\sqrt{n}$ 个节点发生故障，那么存在无故障节点的一行和一列。

**定义** 6.16 (**故障概率** (Failure Probability)). 假设每个节点工作时发生故障 (如宕机) 的概率固定为 $p$(以下假定具体值，例如 $p > 1/2$)。仲裁系统 $\mathcal{S}$ 的故障概率 (Failure Probability) 记为 $F_p(\mathcal{S})$，等于每个仲裁都至少有一个节点发生故障的概率。

**评论:**

- 通常采用 $n \to \infty$ 时的 **渐近故障概率 (Asymptotic Failure Probability)** 来计算 $F_p(\mathcal{S})$。

**事实** 6.17. 根据某个版本的 **切诺夫界 (Chernoff Bound)**，有如下结论：设 $x_1, \ldots, x_n$ 是独立同分布的变量，服从伯努利分布 (Bernoulli-Distributed Random Variables) $Pr[x_i = 1] = p_i$，且 $Pr[x_i = 0] = 1 - p_i = q_i$，于是对 $X := \sum_{i=1}^{n} x_i$ 以及 $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ 有:

$$\text{for all } 0 < \delta < 1\text{: } Pr[X \leqslant (1-\delta)\mu] \leqslant e^{-\mu\delta^2/2} .$$

**定理** 6.18. 过半数仲裁系统 (Majority Quorum System) 的渐近故障概率为 0.

证明. 在过半数仲裁系统中，每个仲裁包含刚好 $\lfloor \frac{n}{2} \rfloor + 1$ 个节点，并且基数 (Cardinality) 为 $\lfloor \frac{n}{2} \rfloor + 1$ 的每个节点集合均构成一个仲裁。若过半数仲裁系统发生故障，则最多只有 $\lfloor \frac{n}{2} \rfloor$ 个节点正常工作。否则，必然存在至少一个仲裁可用。为了计算故障概率，定义以下随机变量：

$$x_i = \begin{cases} 1, & \text{若节点} i \text{ 正常工作, 发生概率为} p \\ 0, & \text{若节点} i \text{ 发生故障, 发生概率为} q = 1 - p \end{cases}$$

并且 $X := \sum_{i=1}^{n} x_i$, 其期望为 $\mu = np$,

此处，$X$ 表示正常工作的节点个数。

为了估计正常工作的节点数目小于 $\lfloor \frac{n}{2} \rfloor + 1$ 的概率，将利用上面的切诺夫不等式。通过选择 $\delta = 1 - \frac{1}{2p}$，我们得到 $F_P(\mathcal{S}) = Pr[X \leqslant \lfloor \frac{n}{2} \rfloor] \leqslant Pr[X \leqslant \frac{n}{2}] = Pr[X \leqslant (1-\delta)\mu]$。

由于 $\delta = 1 - \frac{1}{2p}$，我们有 $0 < \delta \leqslant 1/2$，因为 $1/2 < p \leqslant 1$. 这样，我们可以使用切诺夫界并且得到 $F_P(\mathcal{S}) \leqslant e^{-\mu\delta^2/2} \in e^{-\Omega(n)}$。

**定理** 6.19. 网格仲裁系统 (Grid Quorum System) 的渐近故障概率为 1.

证明. 我们将 $n = d \cdot d$ 个节点布置到一个 $d \times d$ 的网格中。一个仲裁总是包含完整的一行。我们将利用如下伯努利不等式 (Bernoulli Inequality) 不等式：对于任意 $n \in \mathbb{N}, x \geqslant -1 : (1+x)^n \geqslant 1 + nx$。

仲裁系统发生故障，意味着每行中都有至少一个节点发生故障 (对于特定的一行，此概率为 $1 - p^d$，因为该行中所有节点都工作的概率为 $p^d$)。因此，可以用如下式子来限定故障概率：

$F_p(\mathcal{S}) \geqslant Pr[\text{at least one failure per row}] = (1-p^d)^d \geqslant 1 - dp^d \underset{n\to\infty}{\longrightarrow} 1$。[7]

**评论:**

- 现在我们有了一个具有最优负载的仲裁系统 (网格) 和一个具备强大容错能力的仲裁系统 (过半数)，然而如果想兼有这二者该怎么做？

**定义** 6.20 (B-**网格仲裁系统** (B-Grid Quorum System)). 考虑 $n = dhr$ 个节点，布置在 $h \cdot r$ 行、$d$ 列的矩形网格中。每组 $r$ 行构成一个带 (band)，处于同一个带中的一列 $r$ 个单元格称作微列 (mini-column)。一个仲裁由每个带中的一个微列和某个带中的每个微列的一个元素组成；因此每个仲裁有 $d + hr - 1$ 个元素[8]。$B$-网格仲裁系统 (B-Grid Quorum System) 由所有这样的仲裁组成。

**定理** 6.22. B-网格仲裁系统的渐近故障概率为 0。

证明. 假设 $n = dhr$，并且节点被布置在 $d$ 列、$h \cdot r$ 行的网格中。如果在每个带中均有一个完整的微列发生故障，那么 B-网格仲裁系统也必然

[7]译者注：上式最后一步推导中，须注意当 $n \to \infty$ 时，$d \to \infty$，因为 $d = \sqrt{n}$。

[8]译者注：每个带一个微列，加起来就是 $hr$ 个单元格。某个带中每个微列的一个元素，加起来是 $d$ 个单元格。两者必有一个重复。参考图 6.21。

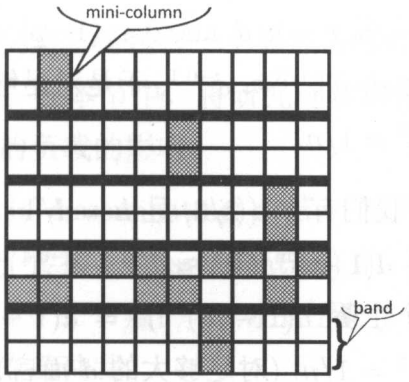



图 6.21: 一个 B-网格仲裁系统，包含 $n = 100$ 个节点, $d = 10$ 列, $h \cdot r = 10$ 行, $h = 5$ 带, 以及 $r = 2$. 图中展示的仲裁包含 $d + hr - 1 = 10 + 5 \cdot 2 - 1 = 19$ 个节点。如果选择均匀的访问策略 $Z$, 则其工作量为 $d + hr - 1$，负载为 $\frac{d+hr-1}{n}$. 若设置 $d = \sqrt{n}$ 且 $r = \log n$, 我们将得到工作量为 $\Theta(\sqrt{n})$ 且负载为 $\Theta(1/\sqrt{n})$。

发生故障，因为不可能挑选出一个带，该带中还存在一个微列，其中每个节点均正常工作。如果在一个带中每个微列均有一个节点发生故障，那么整个系统也发生故障。这些事件可能不是彼此独立的，但是借助于联合上界 (Union Bound[9])，可以用下面的式子来得到故障概率的上界：

$$\begin{aligned} F_p(\mathcal{S}) &\leqslant Pr[\text{每个带中都有一个完整的微列发生故障}] \\ &\quad + Pr[\text{在一个带中，每个微列均有至少一个节点发送故障}] \\ &\leqslant (d(1-p)^r)^h + h(1-p^r)^d \end{aligned}$$

10

[9]译者注：即 $Pr(X) + Pr(Y) \geqslant Pr(X \wedge Y)$。

[10]译者注：上式第一项，一个完整的微列发生故障的概率为 $(1-p)^r$，而一个带中至少一个微列发生故障的概率 $\leqslant d(1-p)^r$，这里用到了作者所说的联合上界，每个带中均有一个微列发生故障的概率则 $\leqslant (d(1-p)^r)^h$。第二项，一个微列至少有一个节点发生故障概率为 $1-p^r$，每个微列均如此的概率为 $(1-p^r)^d$，存在某个带发生这样的情况的概率则 $\leqslant h(1-p^r)^d$ (同样用到联合上界)。

我们使用 $d=\sqrt{n}, r=\ln d$, and $0 \leqslant (1-p) \leqslant 1/3$。由于 $n^{\ln x}=x^{\ln n}$, 我们有 $d(1-p)^r \leqslant d \cdot d^{\ln 1/3} \approx d^{-0.1}$, 于是对足够大的 $d$，整个第一项将小于 $d^{-0.1h} \ll 1/d^2 = 1/n$.

至于第二项，我们有 $p \geqslant 2/3$, 且 $h = d/\ln d < d$。于是我们可以得到该项的一个上界 $d(1-d^{\ln 2/3})^d \approx d(1-d^{-0.4})^d$。由于 $(1+t/n)^n \leqslant e^t$, 我们可以得到一个上界 $d(1-d^{-0.4})^d = d(1-d^{0.6}/d)^d \leqslant d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} = 1/n$. (对足够大的 $d$ 而言)。于是，合在一起，我们就得到了 $F_p(\mathcal{S}) \in O(1/n)$。

| | 单节点 | 多数派 | 网格 | B-网格 * |
|---|---|---|---|---|
| 工作量 | **1** | $> n/2$ | $\Theta(\sqrt{n})$ | $\Theta(\sqrt{n})$ |
| 负载 | 1 | $> 1/2$ | $\Theta(\mathbf{1/\sqrt{n}})$ | $\Theta(\mathbf{1/\sqrt{n}})$ |
| 适应力 | 0 | $< \mathbf{n/2}$ | $\Theta(\sqrt{n})$ | $\Theta(\sqrt{n})$ |
| 错误概率 ** | $1-p$ | $\to \mathbf{0}$ | $\to 1$ | $\to \mathbf{0}$ |

表 6.23: 对比不同的仲裁系统：适应力 (resilience), 工作量 (work), 负载 (load), 以及渐近错误概率。每一行最好的结果用粗体显示。

* 设置 $d=\sqrt{n}$ 且 $r=\log n$

** 假定 $q=(1-p)$ 为常数，且远小于 1/2

## 6.4 拜占庭仲裁系统 (Byzantine Quorum Systems)

尽管宕机节点不好，但它们仍是易于对付的：只需访问所有节点都能够响应的另一个仲裁即可! 然而，拜占庭节点就使得事情更难，因为它们可能假装是一个常规节点，即，需要更精细的方法来对付它们。我

们需要确保两个仲裁的交集总是包含一个非拜占庭节点，此外不允许拜占庭节点渗入每个仲裁。在本节中，将研究逐步增强的三个衡量标准，以及它们对于仲裁系统的负载的影响。

**定义** 6.24 ($f$-**传播** ($f$-disseminating)). 若一个仲裁系统 $\mathcal{S}$ 满足以下两个条件，则称它是$f$-传播 ($f$-disseminating) 的：(1) 任意两个不同仲裁的交集总是包含 $f+1$ 个节点；(2) 对于任意 $f$ 个拜占庭节点，存在至少一个无拜占庭节点的仲裁。

**评论:**

- 根据第 (2) 个条件，即使有 $f$ 个拜占庭节点，它们也不能通过假装已崩溃来停止所有仲裁。至少一个仲裁将存活下来。对于接下来更复杂的拜占庭仲裁系统，我们也将保持这个假设。
- 拜占庭节点也可能作些比崩溃更糟糕的事情-它们可能篡改数据！不过，由于条件 (1)，在每两个仲裁的交集中存在至少一个非拜占庭节点。如果数据是自验证的，比如认证，则这一个好节点就足够了。
- 如果数据不是自验证的，那么需要另一种机制。

**定义** 6.25 ($f$-**掩盖** ($f$-masking)). 若一个仲裁系统 $\mathcal{S}$ 满足以下两个条件，则称它是$f$-掩盖 ($f$-masking) 的：(1) 任意两个不同的仲裁的交集总是包含 $2f+1$ 个节点，并且 (2) 任意 $f$ 个拜占庭节点，存在至少一个无拜占庭节点的仲裁。

**评论:**

- 注意，一个$f$-掩盖仲裁系统与一个$2f$-传播仲裁系统相同。该想法在于非拜占庭节点 (至少 $f+1$ 个) 能以票数胜过拜占庭节点 (至多 $f$ 个)，但是仅当所有的非拜占庭节点都是最新的成立。

- 这带来一个本章迄今尚未涉及的问题：如果访问某个仲裁并且更新其值，该改变仍旧必须要传播到该拜占庭仲裁系统中的其他节点。本节结尾要讨论的不透明仲裁系统 (Opaque Quorum System) 将讨论该问题。
- *$f$-传播仲裁系统*需要超过 $3f$ 个节点，而*$f$-掩盖仲裁系统*需要超过 $4f$ 个节点。实质上，仲裁可以不包含太多的节点，并且不同的交集性质导致不同的性能[11]。

**定理** 6.26. 若 $\mathcal{S}$ 是一个 $f$-传播仲裁系统，则 $L(\mathcal{S}) \geqslant \sqrt{(f+1)/n}$。

**定理** 6.27. 若 $\mathcal{S}$ 是一个 $f$-掩盖仲裁系统，则 $L(\mathcal{S}) \geqslant \sqrt{(2f+1)/n}$。

*对定理 6.26 和 6.27 的证明*. 此处的证明思想类似于定理 6.6的证明，注意现在不是一个来自最小仲裁的节点被访问，而是 $f+1$ 或 $2f+1$ 个。

**定义** 6.28 (**$f$-掩盖网格仲裁系统** ($f$-Masking Grid Quorum System)). 一个 $f$-掩盖网格仲裁系统是一个网格仲裁系统，其中每个仲裁包含完整的一列和 $f+1$ 行节点，并且 $2f+1 \leqslant \sqrt{n}$。

**评论:**

- $f$-掩盖网格的负载几乎达到了 $f$-掩盖仲裁系统负载的下界，但是并未完全达到。下面我们将介绍一个小的改变，并达到渐近最优。

**定义** 6.30 (**$M$-网格仲裁系统** ($M$-Grid Quorum System)). 一个 $M$-掩盖网格仲裁系统也是一个网格仲裁系统，其中每个仲裁包含 $\sqrt{f+1}$ 行以及 $\sqrt{f+1}$ 列，且 $f \leqslant \frac{\sqrt{n}-1}{2}$。

[11] 译者注：负载、工作量等。

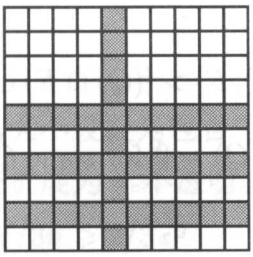


图 6.29: 一个例子，如何在一个 $f$-掩盖的网格中选择一个仲裁，该仲裁须包含 $2+1=3$ ($f=2$) 行节点。当访问策略为均匀时，工作负载为 $\Theta(f/\sqrt{n})$。一个仲裁的列将会和其他仲裁的行相交叉，因此任意两个仲裁重复的节点数至少为 $2f+2$。

**推论** 6.31. $f$-掩盖网格仲裁系统和 $M$-网格仲裁系统都是 $f$-掩盖仲裁系统。

**评论:**

- 我们实现了几乎和无拜占庭节点相同的负载！然而，如前所述，如果访问一个仲裁，除去与其他仲裁的重叠部分外，其他节点都不是最新的。此时此时会发生什么？我们肯定能修复该问题并且不会有太多损失吗？
- 该性质将在本章的最后部分用不透明仲裁系统 (*Opaque* Quorum Systems) 来应对。假设所访问的好的 (保持信息是最新的) 节点数量为 $n_1$，而过期的节点和拜占庭节点数量之和为 $n_2$，该策略能确保 $n_1 > n_2$(参见 (6.32.1))。

**定义** 6.32 ($f$-**不透明仲裁系统** ($f$-Opaque Quorum System)). 若一个仲裁系统 $\mathcal{S}$ 满足以下两个条件，则称其为 $f$-不透明的 ($f$-*opaque*)。对于任意 $f$ 个拜占庭节点集合 $F$，以及任意两个不同的仲裁 $Q_1, Q_2$:

$$|(Q_1 \cap Q_2) \setminus F| > |(Q_2 \cap F) \cup (Q_2 \setminus Q_1)| \tag{6.32.1}$$

$$(F \cap Q) = \emptyset \text{ for some } Q \in \mathcal{S} \tag{6.32.2}$$

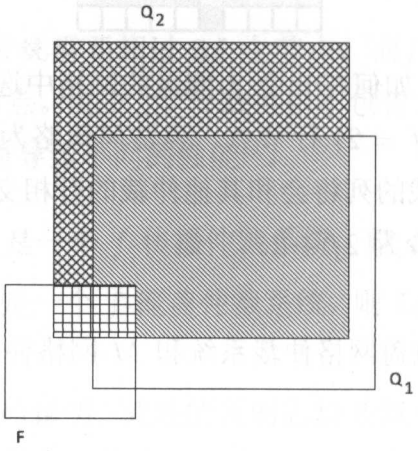



图 6.33: 不透明仲裁系统的交集性质。等式 6.32.1 确保 $Q_1, Q_2$ 的交集中的非拜占庭节点的集合大于过时节点的集合，即使拜占庭节点与那些过时节点“组队”也如此。因此，正确的最新值总是能够被过半数投票识别。

**定理** 6.34. 若 $\mathcal{S}$ 是一个 $f$-不透明的仲裁系统，则 $n > 5f$。

证明. 根据 (6.32.2), 存在一个仲裁 $Q_1$ 其包含节点数至多为 $n - f$。又根据 (6.32.1), 我们有 $|Q_1| > f$。令 $F_1$ 为包含 $f$ (拜占庭) 节点的集合，$F_1 \subset Q_1$, 同时根据 (6.32.2), 存在一个仲裁 $Q_2 \subset V \setminus F_1$。于是，$|Q_1 \cap Q_2| \leqslant n - 2f$。由 (6.32.1), 可得出 $|Q_1 \cap Q_2| > f$。因此，可以挑选 $f$ 个拜占庭节点 $F_2$，使得 $F_2 \subset (Q_1 \cap Q_2)$。依据 (6.32.1)，可以得到：$n-3f > |(Q_2 \cap Q_1)| - |F_2| \geqslant |(Q_2 \cap Q_1) \cup (Q_1 \cap F_2)| \geqslant |F_1| + |F_2| = 2f$。

**评论:**

- 可以将每个仲裁的大小设置为包含 $\lceil(2n+2f)/3\rceil$ 个节点，从而使过半数仲裁系统扩展为 $f$-不透明。此时其负载为：$1/n\lceil(2n+2f)/3\rceil \approx 2/3 + 2f/3n \geqslant 2/3$。
- 我们可以做得更好吗？很遗憾，不能……

**定理** 6.35. 若 $\mathcal{S}$ 是一个 $f$-不透明仲裁系统，则 $L(\mathcal{S}) \geqslant 1/2$。

证明. 从公式 (6.32.1)，可以得出一个结论：对于任意两个仲裁 $Q_1, Q_2 \in \mathcal{S}$, 其交集的大小最多为 $Q_1, Q_2$ 大小的一半, 即：$|(Q_1 \cap Q_2)| \geqslant |Q_1|/2$。令 $\mathcal{S}$ 包含仲裁 $Q_1, Q_2, \ldots$，由一个访问策略 $Z$ 引发的在 $Q_1$ 上的负载为：

$$\sum_{v\in Q_1}\sum_{v\in Q_i} L_Z(Q_i) \geqslant \sum_{Q_i} (|Q_1|/2)\ L_Z(Q_i) = |Q_1|/2\ .$$

利用鸽巢原理 (Pigeonhole Principle)，$Q_1$ 中必然存在至少一个节点，该节点的负载至少为 1/2。

## 章节说明

在历史上，仲裁 (Quorum) 是执行一个组织的业务必须的审议机构成员的最少人数 (如，法定人数)。自 20 世纪 70 年代后期到 20 世纪 80 年代早期，它们的使用启发了将仲裁系统引入到计算机科学中。早期的工作集中于过半数仲裁系统 [Lam78, Gif79, Tho79]，[GB85] 稍后引入了极小性概念。网格仲裁系统最早在 [Mae85] 中出现，在 [NW94] 中介绍了 B-网格。后来的文章和 [PW95] 也开始了负载和适应力方面的研究。

$f$-掩盖网格仲裁系统和不透明仲裁系统来自 [MR98]，而 M-网格仲裁系统是在 [MRW97] 中介绍的。这两篇论文也标志着拜占庭仲裁系统正式研究的开始。$f$-掩盖和 $M$-网格的渐近故障概率为 1，在这些论文中也能找到性能更好的、更复杂的系统。

Quorum 系统也已被扩展来应对节点动态离开和加入，参见 [NW05] 中的动态路径仲裁系统。

对于仲裁系统的进一步综述，可参考 Vukolić 的著作 [Vuk12]，以及 Merideth 和 Reiter 的文章 [MR10]。

# 参考文献

[GB85] Hector Garcia-Molina and Daniel Barbará. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In Michael D. Schroeder and Anita K. Jones, editors, *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Mae85] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR10] Michael G. Merideth and Michael K. Reiter. Selected results from the latest decade of quorum systems research. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.

[MRW97] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 249–257. ACM, 1997.

[NW94] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 214–225. IEEE Computer Society, 1994.

[NW05] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing*, 17(4):311–322, 2005.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[Vuk12] Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus.* Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

# 第 7 章　最终一致性以及比特币

应该怎样实现一个 ATM？下面的实现方法可以正常工作吗？

**算法 7.1** 朴素的 ATM 算法

```
1: ATM 向银行发出取款请求
2: ATM 等待银行的回应
3: if 客户的账户余额充足 then
4:     ATM 输出钞票
5: else
6:     ATM 显示错误
7: end if
```

**评论:**

- 银行与 ATM 之间的通信中断问题可能使算法 7.1 在第 2 行阻塞。
- 网络分区 (Network Partition) 是一类错误，指一个网络分裂为至少两个部分，且这些分裂之后的子网之间彼此不能通信。直觉上，任何非平凡的分布式系统不能在一个分区期间继续工作，且保持一致性。接下来，我们将介绍在一致性 (Consistency)，可用性 (Availability) 和分区容忍性 (Partition Tolerance) 这三个指标间进行取舍的问题。
- 有很多导致分区的原因，比如物理上断开连接，软件错误，或

者是不相容的协议版本。从系统一个节点的角度出发，分区类似于持续一段时间的消息丢失。

## 7.1 一致性、可用性，以及分区

**定义** 7.2 (**一致性** (Consistency)). 一个系统的所有节点就系统的当前状态达成一致。

**定义** 7.3 (**可用性** (Availability)). 系统是可用的且正处理请求。

**定义** 7.4 (**分区容忍性** (Partition Tolerance)). 分区容忍性是指分布式系统具备的一种能力：在存在网络分区的时候仍可以正确地工作。

**定理** 7.5 (CAP **定理**). 一个分布式系统不可能同时实现一致性，可用性以及分区容忍性。它可满足其中任意两个要求，但不能同时满足三个。

证明. 假定两个节点，共享某个状态。两个节点处于不同的分区中，即：它们不能通信。假定一个请求希望更新这个共享状态并联络一个节点来执行更新操作。这个节点可以采取以下两个策略之一：(1) 更新它保存的本地状态，这就导致一个不一致的状态；(2) 不更新它保存的本地状态，那么系统此时就不能响应这个请求，即不可用。

**评论:**

- 算法 7.6[1]是满足分区容忍性和可用性的，因为该算法可以在无法联络银行时也持续处理请求。

[1]译者注：显然，这不是一个安全的，或者说不能保证一致性的算法。ATM 和银行的信息可能不一致，而且，顾客可以在账户没钱的情况下也能取钱。但这个算法确实是满足可用性，以及分区容忍性的。

**算法 7.6** 兼具分区容忍性和可用性的 ATM 算法

```
1: if 可和银行建立连接 then
2:     在银行和 ATM 之间同步账户信息
3:     if 顾客的账户余额不充足 then
4:         ATM 显示错误信息并且退出当前会话
5:     end if
6: end if
7: ATM 输出钞票
8: ATM 记录此次取款记录，以便以后同步
```

- ATM 所存储的账户本地视图 (即 ATM 所看到的账户余额信息) 可能和银行所看到的账户信息不一致，因此该算法不满足一致性。
- 当连接恢复时，算法将会把 ATM 对账户所做的所有改动发送回银行，并和银行同步。这被称为*最终一致性*。

**定义 7.7 (最终一致性** (Eventual Consistency)). 如果不再对共享状态有新的更新，则最终系统进入安静 (Quiescent) 状态，即节点之间不需要发送额外的消息，且共享状态是一致的。

**评论:**

- 最终一致性是*弱一致性*的一种形式。
- 最终一致性保证所有的节点最终就共享状态达成一致，但在某个时间段内，共享状态在各个节点所存的状态可能不一致。
- 分区期间，不同的节点可能执行不同的更新，而这些更新可能在语义上是彼此矛盾的。因此，需要一个*冲突解决*机制来解决这些冲突，并且使得节点最终在一个相同的状态上达成一致。

- 比特币系统就是最终一致性的一个典型例子。

## 7.2 比特币

**定义** 7.8 (**比特币网络** (Bitcoin Network)). 比特币网络是一个随机连接的覆盖网络 (Overlay Network)，它包含成千上万个节点，被各种各样的拥有者控制。所有节点运行相同的操作，即：这是一个去中心化的同质网络 (Homogenous Network)。

**评论:**

- 缺少结构[2]是有意为之的：这保证了一个攻击者不能在网络中特意占据一个关键的位置并且操纵信息交互。信息是通过简单的广播协议来传播的。

**定义** 7.9 (**地址** (Address)). 用户可以产生任意数量的私钥，并基于这些私钥构建一个公钥。地址是基于一个公钥得到的，并且被用来标识比特币系统中一笔金额的接收者。一个公/私钥对被用来唯一地标识某个地址 (以及相应的一笔金额) 的拥有者。

**评论:**

- 术语公钥和地址都是公开的信息，因此经常可以互换使用。使用地址的好处是它比公钥简短。
- 通常很难将地址和拥有该地址的用户关联起来，因此比特币经常被认为是匿名的系统。

[2]译者注：指比特币网络的去中心化，扁平化。

- 不是每个用户都需要运行一个带完整验证功能的节点，很多终端用户倾向于使用一个轻型的客户端，这些客户端只是临时与网络相连。
- 比特币网络以协作的方式来追踪每个地址的比特币余额。
- 地址包含一个网络标识字节，以及公钥的哈希码和校验和。通常以 base58 的编码存储。这是一个类似于 base64 的编码，但去掉了一些容易混淆的字符。比如 base58 去掉了小写字母“l”, 因为它和数字“1”很相似。
- 哈希算法将得到长度为 20 字节的的地址。这意味着总共可以有 $2^{160}$ 个不同的地址。如果用穷举算法来破解一个地址，就算每秒尝试 10 亿次，大约也需要 $2^{45}$ 年才可以找到一个匹配的公/私钥密码对。根据生日悖论 (Birthday Paradox)，如果我们不是去穷举某一个特定地址而是穷举任意地址，猜中的几率将会上升。然而，虽然随着所尝试穷举地址数目增多，成功的几率会增大，尝试的成本也会随之上升。

**定义 7.10 (输出 (Output))**. 一个输出是一个元组，包含一定数额的比特币以及一个使用条件。绝大多数情况下，使用条件需要一个和某地址对应私钥相关联的有效签名。

**评论:**

- 使用条件是一段脚本，包含多个选项。除了一个签名外，脚本还可以要求一个简单计算的输出结果，或者是一个密码学难题 (Cryptographic Puzzle) 的答案。
- 输出存在两个状态：未使用 (Unspent) 或者已使用 (Spent)。任何输出只能被使用一次。一个地址的账户余额是所有与该地址关联的未使用输出的比特币数额总和。

- 所有未使用的交易输出 (Unspent Transaction Outputs, UTXO) 以及一些附加的全局参数就构成了比特币网络的共享状态。每个在比特币网络中的节点都拥有一个该状态的一个完整副本。这些本地副本之间可能暂时性地不一致，但是最终将重新达成一致性。

**定义 7.11 (输入 (Input))**. 一个输入是一个元组，包含对前面一个已经创建的输出的引用，以及用于该输出中使用条件的一组参数 (签名)。这些参数将证明交易创建者有权使用所引用的输出。

**定义 7.12 (交易 (Transaction))**. 交易是一个数据结构，描述了一次比特币的转移 (使用者到接收者) 情况。一个交易包含很多输入和新创建的输出。这些输入将导致所引用的输出变为已使用 (即：从 UTXO 中删除)，此外新创建的输出将被增加到 UTXO 中。

**评论:**

- 输入用一个 $(h, i)$ 元组来引用一个即将被使用的输出，此处 $h$ 是创建该输出的交易的哈希值，而 $i$ 描述了在交易中该输出的索引值。
- 交易在比特币网络中广播，网络中每个接收到该交易的节点都需要处理它。

**评论:**

- 须注意一个交易对共享状态 (定义 7.10后的评论) 的效果是确定的。换句话说，如果所有的节点按相同的顺序接收到相同的交易 (参见定义 2.8)，则在该共享状态在所有节点之间将保持一致。

**算法 7.13** 节点处理接收到的交易

```
接收到交易 t
for each t 中的输入 (h, i) do
    if 输出 (h, i) 不在本地的 UTXO or 签名无效 then
        将交易 t 删除掉，并停止处理此交易
    end if
end for
if 所有输入包含的金额之和 < 所有新创建的输出的金额之和 then
    将交易 t 删除掉，并停止处理此交易
end if
for each t 中的输入 (h, i) do
    从本地 UTXO 中删除 (h, i)
end for
将 t 添加到本地历史
将 t 发送给比特币网络中的邻居
```

- 一个交易产生的输出，其包含的总金额可能比所有输入的总金额要少。此时，输入和输出总金额之间的差额成为交易费。交易费被用来激励系统的其他参与者 (参见定义 7.19)。
- 注意到到现在为止，我们仅仅描述了一个局部的接受策略。还没有讨论任何措施来避免：不同的节点接受不同的交易，但这些交易却使用同一个输出。
- 交易将处于以下两个状态之一：未确认 (Unconfirmed) 或已确认 (Confirmed)。从广播中接收到所有交易是未确认的，并且会被加入到一个名为记忆池的交易池中去。

**定义 7.14 (重复使用** (Doublespend)). 重复使用指一个特殊状况，此时多个交易都尝试使用同一个输出。只有一个交易可以是有效的，因

为输出只能被使用一次。当节点在重复使用的情形下接受不同的交易时，共享状态将变得不一致。

**评论:**

- 重复使用可以很自然地出现，比如，当输出被多个用户共同拥有的时候。然而，很多时候重复使用是故意的行为——这被称为重复使用攻击。在一个交易中，一个攻击者假装将一个输出转移给一个受害者，而实际只是重复使用在另一交易中的同一个输出，在那个交易中，该输出会被转回给攻击者自己。
- 重复使用可以导致不一致的状态，因为一组交易的有效性取决于它们到达的顺序。如果两个相互冲突的交易被同一个节点看到，该节点将认为第一个是有效的 (参见算法 7.13)，并认为第二个交易是无效的，因为该交易尝试使用已经被用过的输出。然而，不同的节点收到交易的顺序可能是不一致的，因此共享状态也就不一致。
- 如果重复使用问题没有被解决，共享状态将出现分叉。于是就需要一个冲突解决机制来判定冲突的交易中，哪一个交易应该被确认 (即：被所有节点接受)，由此实现最终一致性。

**定义 7.15 (工作量证明 (Proof-of-Work)).** 工作量证明 (Proof-of-Work, PoW) 机制使得一个参与者可以向其他参与者证明：他已在一段时间内持续使用了一定数量的计算资源。特别地，定义工作量证明函数 $\mathcal{F}_d(c,x) \to \{true, false\}$，此处 $d$ 是一个正实数，表明困难程度；挑战问题 $c$ 和随机数 $x$ 通常是比特字符串。工作量证明函数需要具备下面的性质：

1. 当 $d, c, x$ 都给定时，很快就能计算出 $\mathcal{F}_d(c,x)$。

2. 对一组给定的参数 $d$ 和 $c$，找到 $x$ 使得 $\mathcal{F}_d(c,x) = true$ 是可计算的，但是非常困难。困难系数 $d$ 被用来调节找到 $x$ 的平均期望时间。

**定义** 7.16 (**比特币** PoW **函数**). 比特币 PoW 函数定义如下：

$$\mathcal{F}_d(c,x) \to \text{SHA256}(\text{SHA256}(c|x)) < \frac{2^{224}}{d}.$$

**评论:**

- 此函数将挑战 $c$ 和随机数 $x$ 连起来，并且使用 SHA256 做两次哈希。SHA256 的输出是一个密码学的哈希值，取值范围是 $\{0,\ldots,2^{256}-1\}$。将这个输出值和目标值 $\frac{2^{224}}{d}$ 进行比较。增大难度系数 $d$，目标值将减小，从而提高了找到 $x$ 的难度。
- SHA256 是一个用于加密的哈希函数，其输出是伪随机的。目前还没有比穷举更好的算法来寻找 $x$ 使其满足函数 $\mathcal{F}_d(c,x)$。从设计的层面来讲，这使得找到 $x$ 很困难，但是却很容易验证一个 $x$ 是否合乎要求。
- 如果所有计算 PoW 函数的节点使用相同的挑战，那么计算能力最强的节点将总是获胜。然而，就如我们即将在定义 7.19中所看到的，每个节点都使用一个节点特定的挑战来尝试找到合理的随机数 x。

**定义** 7.17 (**区块** (Block)). 一个区块是一个数据结构，在一个节点局部状态之上累积的改变将打包在区块中并传递给整个网络。一个区块包含了一组交易，一个指向上一个节点的引用，和一个随机数 (即在工作量证明阶段找到的随机数)。一个区块包含了本区块创建者 (矿工) 所接受并存放在自己记忆池中的所有交易，这些交易都是在上一个区

块之后产生的。一个节点在找到一个有效的随机数来满足它的工作量证明函数后，将广播一个区块。

**算法 7.18** 节点寻找区块

1: $x = 0$, $c$, $d$, 上一个区块 $b_{t-1}$
2: **repeat**
3: $\quad x = x + 1$
4: **until** $\mathcal{F}_d(c, x) = true$
5: 广播区块 $b_t = (memory\text{-}pool, b_{t-1}, x)$

**评论:**

- 由于每个区块都引用它前面的那个区块，所有的区块就构成了一棵树，以创世区块为树根。
- 使用工作量证明机制的首要目的是调节整个网络找到区块的速度，使得网络有时间来实现在最新一个区块上的同步。比特币通过设置难度系数来确保整个网络找到一个新的区块的平均时间为 10 分钟。
- 找到一个区块使得发现者可以把自己记忆池中的所有交易强加给所有其他节点。在接收到一个区块的时候，所有的节点都将回滚自己在上一个区块之后对本地状态所做的任何改动，并执行新区块包含的交易。
- 我们称一个区块内包含的交易被该区块确认。

**定义** 7.19 (**奖励交易** (Reward Transaction)). 在一个区块内的第一个交易被称为奖励交易 (Reward Transaction)。发现该区块的矿工将获得一定数量的新比特币以奖励它确认了一组交易。奖励交易有一个名

义上的输入，其输出的总和包括一个定额的补贴加上被该区块所确认的所有交易的交易费之和。

**评论:**

- 比特币系统的交易有一个规则：所有输入之和必须大于或等于所有输出之和。但*奖励交易*是唯一的意外。
- 每个*奖励交易*所创造的比特币数量 (这些比特币会被奖励给相应的矿工) 由一个补贴机制来决定，该机制是比特币网络协议的一部分。最初，每个区块包含的补贴为 50 个比特币，这个数量在每找到 210,000 个区块后 (或者是大约每 4 年) 之后就会减半。依据这个机制，能流通的比特币总量不会超过 2100 万。
- 设计者们期望，找到一个区块的代价，比如能源开销及其他成本，应该和矿工得到的奖励价值接近。

**定义** 7.20 (Blockchain). 从创世区块开始，到某个叶节点为止的最长路径，被称为区块链。区块链的作用是维护一个一致的交易历史，而所有的节点最终都将在唯一的区块链 (交易历史) 上达成一致。

**评论:**

- 从创世区块到某个区块 $b$ 的路径长度是该区块的高度 $h_b$。
- 只有从创世区块到某个叶节点所构成的最长路径才是一个有效的交易历史。注意到由于*重复使用*的存在，分支之间可能会彼此冲突。
- 由于只有在最长路径上的交易才会被最终承认，矿工们都会自发地将它们的区块加到最长的链上，这样就会在当前状态上达成一致。

- 挖矿的动力迅速提升了工作量证明机制的难度。最初，矿工们使用 CPU 来寻找区块，但是很快就更换为了 GPU，FPGA，甚至是应用特定的集成电路 (AS-ICs)。这导致了目前的局面：只有最高效的矿工，采用最快的硬件和最便宜的电力，才有机会盈利。
- 如果多个区块同时 (或近乎同时) 被找到，系统将出现分叉。分叉是很自然的，因为挖矿是一个分布式的随机过程，而两个区块是可能在几乎同一时刻被找到。

**算法 7.21** 节点接收到区块

1: 接收到区块 $b$
2: 当前节点所保存的头区块为 $b_{max}$，其高度为 $h_{max}$
3: 将区块 $b$ 加到树上去，作为其父区块 $p$ 的子区块。新增区块的高度为 $h_b = h_p + 1$
4: **if** $h_b > h_{max}$ **then**
5: $\quad h_{max} = h_b$
6: $\quad b_{max} = b$
7: $\quad$ 根据从创世区块到新增区块的路径来计算 UTXO
8: $\quad$ 清空记忆池
9: **end if**

**评论:**

- 算法 7.21 描述了一个节点在接收到一个区块时是如何更新它本地状态的。注意到，类似于算法 7.13，这只是本地的策略，而节点之间仍然可能出现状态分歧，也就是说接受不同的区块作为相同高度的头区块。

- 与扩展当前路径不同，切换路径可能导致已经被确认的交易重新变为“未确认”，因为在新的路径上的区块没有包含这些交易。切换路径被称为重组 (reorg)。
- 清空记忆池包括：(1) 删除在当前路径上已经被确认的交易；(2) 删除与已经被确认的交易存在冲突的交易；(3) 增加在前面路径中已被确认，但是在当前路径不再被确认的交易。
- 为了避免在每个新节点加入时重新计算整个 UTXO，所有当前的实现都使用数据结构来保存了一个区块所执行的撤销 (undo) 信息。这使得我们可以方便地在路径上移动，从而快速地切换路径，并更改头区块。

**定理** 7.22. 分叉将最终被解决，并且所有节点最终都将接受同一条最长路径。于是系统确保了最终一致性。

证明. 如果分叉持续存在，在两个分支上都需要不断产生新的区块，且它们的高度还得保持一致，否则较短分支上的节点将切换到较长的分支上去。随着分叉长度的增加，两个分支同时找到新节点的概率将呈指数形式下降。于是最终，将出现一个时刻，只有一个分支会被扩展并成为最长的分支。

## 7.3 智能合约 (Smart Contracts)

**定义** 7.23 (**智能合约** (Smart Contract)). 一个智能合约是在两个或多个参与者之间达成的协定，其实现方式使得区块链能自动确保该协定正确执行。

**评论:**

- 合约使得商业逻辑以比特币交易的形式编码，这样就可以确保合约参与者们所达成协定的行动会被执行。如果有参与者没有履行协定，区块链将扮演一个冲突调解者的角色。
- 在比特币的交易中，输出的使用条件 (Spending Condition) 是用脚本来描述的，这使得智能合约成为可能。脚本，以及时钟之类的额外条件，可以共同构成复杂的条件，描述了谁、在什么时候, 能使用一个输出所绑定的金额。

**定义 7.24 (时钟** (Timelock)). 比特币提供了时钟这个机制来使得交易只有到了未来某个时刻才能生效。一个交易可以规定一个时钟，这可以是一个 Unix 系统时间戳，也可以是区块链的高度，描述了何时可以将该交易纳入到某个区块并且确认。

**评论:**

- 设置了时钟的交易将不会被发布到网络中，直到时钟期满失效。接收到该交易的节点有责任将该交易存储在本地，直到该时钟期满失效后才将此交易发布到网络中。
- 包含尚未到期时钟的交易[3]是无效的。区块将不会收录这样的交易。若一个区块包含了尚未到期的交易，该区块是无效区块。节点一旦收到无效的区块或交易，就会立即将它们丢弃并且不会将它们转发给其他节点。
- 时钟可以用来实现交易的替代或接替。假设两个交易 $t_0$ 和 $t_1$，分别包含时钟 $c_0$ 和 $c_1$，且 $c_0 < c_1$。两个交易都企图使用同一

[3]译者注：后文也称为“尚未到期的交易”。

个输出。由于 $t_0$ 的时钟靠前，它将先于 $t_1$ 生效并在网络上广播。因此，交易 $t_1$ 被 $t_0$ 替代。

**定义 7.25 (单签名和多签名的输出** (Singlesig and Multisig Outputs)). 若一个输出使用单一的签名，该输出被称为单签名输出 (singlesig)。相应的，多签名输出的脚本描述了 $m$ 个公钥并且要求与之匹配的 $k$ 个 $(k \leqslant m)$ 个有效签名，这 $k$ 个签名须匹配不同的公钥。

**评论:**

- 绝大多数智能合约都以创建一个 2-of-2 的多签名输出开始，并要求每个参与者提供一个有效的签名。一旦创建这个多签名输出的交易被区块链确认，参与者们都可以确保该输出所附带的金额不可能被单方面使用。

**算法 7.26** 参与者 $A$ 和 $B$ 创建一个 2-of-2 的多签名输出 $o$

1: $B$ 向 $A$ 发送一组输入 $I_B$，其中包含的比特币总额为 $c_B$
2: $A$ 选择他自己的输入集 $I_A$，其中包含的比特币总额为 $c_A$
3: $A$ 创建一个交易 $t_s\{[I_A, I_B], [o = c_A + c_B \rightarrow (A, B)]\}$
4: $A$ 创建一个带时钟的交易 $t_r\{[o], [c_A \rightarrow A, c_B \rightarrow B]\}$ 并对其签名
5: $A$ 将 $t_s$ 和 $t_r$ 发送给 $B$
6: $B$ 对两个交易 $t_s$ 和 $t_r$ 进行签名并且将它们发送给 $A$
7: $A$ 对 $t_s$ 签名并且将其广播到比特币网络

**评论:**

- $t_s$ 被称为准备交易 (Setup Transaction)，其目的是将交易涉及的资金锁到一个共享的账户中去。当 $t_s$ 被签名并且发布到网络中之后，若某个参与者不能履行协定，则这笔多签名输出将不

能被使用。这样会导致这笔资金不能被使用。为了避免出现这样的情形，上述协议创建了一个带有时钟的退款交易 (Refund Transaction)$t_r$，如果在时钟到期之前，协定的交易没有成功，交易 $t_r$ 将确保让资金退回给各位参与者。在任何时刻，只要某个参与者拥有一个带有完整签名的准备交易，另一个参与者必定会拥有一个带有完整签名的退款交易。这确保了资金能最终被退回。

- 两个交易都要求参与者的共同签名。对于准备交易而言，该交易包含来自两个参与者的输入 $I_A$ 和 $I_B$，这要求各自的签名。对于退款交易来说，其输入 $o$ 将使用的是一个多签名的输出。

**算法 7.27** 简单的微额支付通道 (Micropayment Channel) (从 $S$ 到 $R$，容量为 $c$)

1: $c_S = c$, $c_R = 0$
2: $S$ 和 $R$ 采用算法 7.26 来设置输出 $o$，其中包含来自 $S$ 的金额 $c$
3: 创建一个结算交易 (Settlement Transaction) $t_f\{[o], [c_S \to S, c_R \to R]\}$
4: **while** 通道是开放的 **and** $c_R < c$ **do**
5: 交易商品，价值为 $\delta$
6: $c_R = c_R + \delta$
7: $c_S = c_S - \delta$
8: 更新 $t_f$，修改其输出 $[c_R \to R, c_S \to S]$
9: $S$ 对 $t_f$ 签名并发送给 $R$
10: **end while**
11: $R$ 对最后一个 $t_f$ 签名并将其广播到网络

**评论:**

- 算法 7.27 实现了一个简单的*微额支付通道* (Micropayment Channel)，或者说一个智能合约。该合约可以用来快速调整从一个使用者到接收者之间的支付金额。只有两个交易需要在区块链上广播，并加入到区块链上去，即：*准备交易* $t_s$ 以及最后一个*结算交易* $t_f$。在这中间，可以对*结算交易* $t_f$ 做任意次更改，将更多的金额从共享账户转移到接收者那里去。
- 用来创建通道的比特币金额 $c$ 也是在这个简单的*微额支付通道*上可以被转移的最大金额。
- 接收者 $R$ 在任意一个时间点都能确保最终收到 (发送者所承诺支付的) 比特币，因为他总是拥有一个具备完整签名的*结算交易*[4]。然而消费者 $S$ 只拥有一个具备部分签名的“结算交易”。
- 这个简单的*微额支付通道*在本质上是单向的。接收者可以选择在协议执行过程中任意一个*结算交易*[5]，因此接收者可以选择支付金额最大的那个*结算交易*。假设在协议执行过程中，需要减少一部分支付金额，消费者 $S$ 可以提交一个交易来取回部分已支付的比特币，但接收者 $R$ 可以选择不广播这个交易。

## 7.4 弱一致性

最终一致性只是弱一致性的一种形式。在过往的文献中，有很多对分区容忍性和一致性的权衡方案。

**定义** 7.28 (**单调读一致性** (Monotonic Read Consistency)). 如果一个节点 $u$ “看到”了某对象的一个值，后续对 $u$ 的访问都不会返回任

[4]译者注：算法第 9 行。

[5]译者注：每次更改*结算交易*后，消费者 $S$ 都需要对其签名并发送给接收者。

何旧的值。

**评论:**

- 在一个在线的社交网络中，一个很糟糕的场景是：用户收到系统提示，有人对他发表了评论，但他却发现自己不能对这条评论进行回复，因为 Web 界面还没有显示这条评论。在这个例子中，“提示”可以看作是第一个读操作，而在 Web 界面上查阅评论可以被看作是第二次读操作。

**定义 7.29 (单调写一致性** (Monotonic Write Consistency)). 一个节点对某个数据项的写操作必须在该节点任何后续的写操作之前完成 (也就是说，系统保证同一个节点的多个写操作是序列化的)。

**评论:**

- ATM 必须将所有的操作按原顺序重现，否则就可能出现一个较早的操作重写一个靠后的操作结果，这将导致一个不一致的最终状态。

**定义 7.30 (读自身写一致性** (Read-Your-Write Consistency)). 当一个节点 $u$ 已经更新某个数据项之后，该节点发出的所有后续读操作将不会看见旧的值。

**定义 7.31 (因果关系** (Causal Relation)). 下面的成对操作被称为因果关联 (Causally Related) 的：

- 同一个节点对不同变量的两个写操作。
- 同一个节点上，一个读操作之后接着一个写操作。

- 一个读操作，其读取的值是某节点的一个写操作的结果。
- 上述定义的因果关联关系是传递的，两个操作可以通过多个相连的因果关联关系而产生因果关联。

评论:

- 第一个规则确保同一个节点的多个写操作总是保持同样的顺序。例如，如果一个节点 $u$ 在某变量 $x$ 上写了一个值，紧接着该节点通过写另一个变量的值来发出信号：已经更改了 $x$ 的值。如果这两个写操作没有因果关联，另一个节点 $v$ 可能看到了第二次写操作，知道 $u$ 更改了 $x$ 的值，但是 $v$ 依然读取了 $x$ 原先 (未更改前) 的值。

**定义 7.32 (因果一致性** (Causal Consistency)). 在一个系统中，如果有潜在因果关联的操作都以相同的顺序被系统中每个节点看到，则称该系统提供了因果一致性。并发的写操作不是因果关联的，可能被不同的节点以不同的顺序看到。

## 章节说明

CAP 定理首先被 Fox 和 Brewer 提出 [FB99]，然而很多时候人们会认为这个定理是 Eric Brewer 在 2000 年的一次报告中提出的 [Bre00]。稍后，Gilbert 和 Lynch 在异步模型下证明了该定理 [GL02]。Gilbert 和 Linch 也描述了如何在一个部分同步的系统中放宽一致性要求，以实现可用性和分区容忍性。

比特币首先由 Satoshi Nakamoto (中本聪) 在 2008 年提出 [Nak08]。有人认为 Nakamoto 其实是某人或者某个组织的笔名，关于比特币的发明者有很多推测甚至阴谋论，至今仍无定论。

这些猜测包括了密码学家 Nick Szabo [Big13] 以及 Hal Finney [Gre14]。在 Nakamoto 的论文发表之后，很快就出现了第一个比特币客户端，而第一个区块在 2009 年 1 月 3 日发布。这个创世区块包含了当天泰晤士报的头条“The Times 03/Jan/2009 Chancellor on brink of second bailout for banks”(2009 年 1 月 3 日泰晤士报财政大臣即将向银行提供第二次紧急援助)。包含这个头条消息的目的是证明该创世区块确实是在那一天挖掘出来的，并且在那天之前没有人曾发现这个区块。而这也被认为是一个暗示：比特币创建于一场由于银行业不善管理带来的金融危机之中。比特币迅速在无政府主义者和自由主义者中间流行开来。最初的这个客户端如今被一群独立的核心开发人员管理，一直都是比特币网络中最常用的客户端。

比特币的核心是解决由于重复使用带来的冲突，其办法是等待交易被区块链接受。该解决方案的代价是确认一个支付需要很大的延迟时间，但是在很多场景中，我们却需要迅捷的支付手段。

Karame 等人 [KAC12] 展示了接受未被确认的交易将导致一个严重的后果：比特币网络被一个重复使用攻击成功欺骗的可能性将增加到不可忽略的程度。这个问题可以通过所谓信息衰减 (I nformation Eclipsing) [DW13] 的办法来缓解：节点不转发有冲突的交易，因此受害者将不会看见重复使用的两个交易。Bamert 等人 [BDE+13] 展示了另一个改进方案，通过连接大量的节点并追踪交易在网络上的传播过程可以提高发现重复使用攻击的几率。

比特币也不具备很好的可扩展性，因为它依赖于区块链的验证机制。每个节点都需要保存完整的交易历史以引导新加入的节点，这就要求从创世区块开始重建完整的交易历史。简单的微额支付通道是由 Hearn 和 Spilman [HS12] 提出的，它可以被用来将两个参与者之间多笔资金转移捆绑在一起，前提是这些资金转移都限制在该通道创建时

所锁定的资金[6]。

近来，Duplex 微额支付通道 [DW15] 以及闪电网络 [PD15] 相继被提出，以建立双向的微额支付通道。在一个双向微额支付通道中，资金可以向两个方向做任意次转移。这就极大地提升了比特币转移的灵活性，也带来了一些新的功能：例如在两个端点之间的微额支付或者是支付路径。

# 参考文献

[BDE+13] Tobias Bamert, Christian Decker, Lennart Elsen, Samuel Welten, and Roger Wattenhofer. Have a snack, pay with bitcoin. In *IEEE Internation Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, 2013.

[Big13] John Biggs. Who is the real satoshi nakamoto? one researcher may have found the answer. http://on.tcrn.ch/l/R0vA, 2013.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Chan-

[6]译者注：参见算法 7.27及后面的评论。

nels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[FB99] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems*. IEEE, 1999.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[Gre14] Andy Greenberg. Nakamoto's neighbor: My hunt for bitcoin's creator led to a paralyzed crypto genius. http://onforb.es/1rvyecq, 2014.

[HS12] Mike Hearn and Jeremy Spilman. Contract: Rapidly adjusting micro-payments. https://en.bitcoin.it/wiki/Contract, 2012. Last accessed on November 11, 2015.

[KAC12] G.O. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. In *Conference on Computer and Communication Security*, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

# 延伸阅读：PoW vs. BFT[1]

刚接触区块链的人常有这样的疑问：区块链系统一定需要像比特币网络那样挖矿吗？拜占庭容错和挖矿是什么关系？本章将尝试将两种共识机制进行对比，即：比特币挖矿 (准确地说，是 Proof-of-Work，PoW) 以及拜占庭容错 (Byzantine Fault Tolerance，BFT)。本章为译者增加的内容，旨在让读者更深入地了解常见的挖矿机制和本书主要讨论的拜占庭共识之间的区别和联系。本章所讨论的 PoW 默认是比特币网络的工作量证明机制。

PoW 和 BFT 目的都是在去中心化且不可信的分布式环境中达成共识，但两者思路不同，所能达成的共识在效果上存在细微差别，而在性能 (吞吐率和可扩展性) 上面则呈现两极分化。

PoW，可以认为是间接达成共识。它的做法是网络的参与者都在努力解决一个数学难题，该题目很难得到答案 (且只能用类似穷举的方法来求解)，但很容易验证答案是否正确。一旦某个参与者得到一个正确解，它就立即向整个网络广播这个答案，并发布一个新的区块。这个获胜的节点可以决定新区块的内容，但是必须符合一定的规则。

一般情况下，这个新的区块将成为整个网络的共识。只有在发生分叉的情况下，某些区块才可能被抛弃。PoW 的做法类似于现实社会中我们选举出来一位领导人，然后领导人在遵从法律的前提下，做出决定。领导人所做的决定就成为整个社会的共识。也就是说，网络的参与

[1]译者注：本章为译者编写。

者们并非直接决定共识的具体内容，而是通过 PoW 来竞争记账权，由*竞争获胜者来设置共识的内容。因此，我们将其称为间接地达成共识。*

BFT 则可认为是直接达成共识。与 PoW 不同，BFT 不需要先选出优胜者。网络的参与者直接通过投票的方式来决定共识的内容。一旦达成一致，新的共识就会被加入到区块链上去。在 BFT 中，达成一致的共识将不会被丢弃。这也是 BFT 比 PoW 效率更高的一个重要原因。如果用现实社会做例子，BFT 有点像股东大会。所有股东直接投票决定公司应采取什么经营策略。

下面我们首先分析两者在共识层面的区别。首先，我们讨论共识的不可改变性 (Consensus Finality)。从区块链的角度来讲，共识的不可改变性要求：若一个区块被加到区块链上，则它永远不会被移除，也不会被改变位置。换句话说，若某个好节点在区块链上增加了一个区块 $b$，则不会有另一个好节点增加另一个区块 $b'$，使得 $b'$ 的位置在 $b$ 之前。PoW 显然是不满足这个性质的。多个参与节点可能在几乎同一时间内得到正确解，并在网络上发布。因此比特币区块链存在分叉的可能，通常分叉是通过所谓最长链机制来解决。因此，就算某个节点解出了难题，并发布了一个新的区块。该区块也可能最终被网络抛弃。因此，比特币网络通常需要等待 6 个区块才能以**很高的概率**确认某个交易被网络接受。这是 PoW 和 BFT 的一个本质区别。

共识的不可改变性是 BFT 机制的基本要求。事实上，在区块链之前，BFT 最常见的应用是状态机复制 (State Machine Replication)，用来确保多个数据库副本和主本保持一致。共识的不可改变性实质上是要求每条共识 (在状态机复制里通常称为命令) 在各个副本上按相同的顺序执行。BFT 无须等待多个区块才去确认交易，因此效率可以大幅度提高。而且一旦在网络中达成共识，相应的交易就明确无疑地 (Deterministic) 被整个网络确认。

此外，PoW 机制无须任何身份认证。本质上，只要一个节点具备足够的算力，能够抢先算出一个正确解就可以发布一个区块，并赢取相应的奖励。加入比特币区块链网络非常简单，只需要下载并运行相应的代码，连接上网络中任何一个已知的节点就可以。这也是比特币区块链通常被称为公有链的原因。相反地，BFT 协议要求每个节点知道网络中所有其他节点，因此在网络建立的初始阶段，需要一个可信的节点发布各个节点的身份以及数字证书。这也是采用 BFT 的区块链通常被称为私有链的原因。

接下来，我们分析 PoW 和 BFT 在性能上面的区别，包括响应时间、吞吐率，以及可扩展性。

我们所讨论的响应时间，指一个交易从提出到被网络确认平均所花费的时间。以比特币为代表的 PoW 在这方面存在天然的局限性。当一个优胜者产生时，它必须让成千上万个节点知晓这一信息并采取必要的动作，例如验证新区块内容的有效性，同步本地的区块记录等。信息的广播和节点的操作都需要时间，因此新区块的产生不能过于频繁。

事实上，PoW 有两个最基本的参数，即：计算一个区块平均所需时间，以及一个区块的大小。其中第一个参数目前被设置为 10 分钟，这可以通过对难度系数的调整来实现。此外，PoW 只能通过连续的几个区块确认才能近似满足共识的不可改变性，这导致一个交易最终被整个网络确认需要 6 个区块，也就是平均 60 分钟。这样的响应时间无法满足在线交易系统的要求。与之相比，BFT 的性能很占优势，其响应时间基本只受限于网络通信延迟。特别是目前的 BFT 算法大都具备所谓 Early-Stopping 性质，即当不存在拜占庭节点时，整个网络将很快达成共识。同时，BFT 也不需要多个区块来确认交易。

在吞吐率上，PoW 也处于下风。如前所述，平均 10 分钟才能产生

一个区块。而每个区块的大小也不能随意增加。若增大区块的大小，则一个区块在网络中传输所花的时间就会更长，网络产生分叉的机会将会增加，这会带来安全上的风险。根据目前比特币网络的情况，一个区块的大小为 2MB，若一个交易的评价大小为 512 字节，则一个区块平均可容纳 4000 个交易。也就是说，平均 10 分钟处理 4000 个交易，每秒约 7 个交易。这样的吞吐率显然是非常低效的。而 BFT 的吞吐率可以做到每秒 10,000 个左右。

PoW 的优势体现在可扩展性上，比特币网络包含成千上万个节点。而 BFT 的网络规模则相对小很多。BFT 的传统应用，数据库副本备份，通常只有几十个节点。目前文献里看到的 BFT 应用，网络节点数一般也就几百个。虽然没有理论证明，BFT 可以支撑多少个节点，但我们还没有看到有大规模 BFT 网络的出现。

根据 CAP 理论，一个分布式系统只能在一致性、可用性、分区容忍性三项指标中满足两项。PoW 机制采取的是弱一致性，也就是放宽了对一致性的要求，因此可以获得可用性和分区容忍性。而 BFT 满足的是严格的一致性，在同样满足可用性的情况下，只能在分区容忍性上做出让步。

当然，PoW 机制还有一个很受诟病的缺陷，需要消耗大量的能源来求解密码学难题。目前全世界比特币挖坑的能源消耗在 10GW 左右。在全球气候变暖，以及社会对空气污染非常关注的今天，如此巨大的能源消耗也让比特币系统受到了很多批评。与之相比，BFT 网络所耗费的能源则微不足道。

未来的发展趋势：两种机制有很大机会相互融合，取长补短。但是，受限于 CAP 理论，我们不可能同时追求一致性、可用性和分区容忍性。因此，只能根据应用做出适当的剪裁，量体裁衣，设计合适的算法，以

最大程度地满足应用需求。

# 第 8 章　分布式存储

如何将每个大小为 1GB 的 100 万 (1M) 部电影，存储在 100 万个节点上 (每个节点的存储空间为 1TB)？一个简单的办法，我们随意地将这些电影存储在不同的节点上，并采用一个全局的索引记录每个电影的存放位置。但如果电影或节点频繁改变，但我们又不希望频繁地修改这个全局索引，还有什么好的办法吗？

## 8.1　一致性哈希 (Consistent Hashing)

有不少经典哈希算法的变种可以解决这个问题，比如一致性哈希 (Consistent Hashing)：

**算法 8.1** 一致性哈希

1: 采用一组不同的哈希函数，根据每部电影 $x$ 的文件名称分别计算出一组哈希值 $h_i(m) \to [0,1), i = 1, \ldots, k$

2: 采用同样一组哈希函数 $h_i$，根据每个节点 $u$ 的唯一名称 (比如 IP 地址和端口号)，分别计算一组哈希值 $h_i(u) \to [0,1), i = 1, \ldots, k$

3: 对一个电影 $x$，若 $h_i(x) \approx h_i(u)$，则将其的一个副本存放在 $u$ 上。具体而言，将电影 $x$ 存放于节点 $u$ 上，若下面的条件成立：

$$|h_i(x) - h_i(u)| = \min_u\{|h_i(x) - h_i(u)|\}, \text{对任意的} i$$

**定理** 8.2 (**一致性哈希** (Consistent Hashing)). 根据算法 8.1的策略，则每个节点存放的电影数的平均期望为 $km/n$。

证明. 对于某个电影和某个哈希函数，所有的 $n$ 个节点都有同样的机会 $1/n$ 来存储该电影 (即：该节点的哈希值和电影的哈希值最接近)。考虑到有 $m$ 部电影，$k$ 个哈希函数，由于期望的线性[1]，我们可以得到每个节点存放的电影数量的期望为 $km/n$。

**评论:**

- 我们可以做一个简单的计算。现在有 $m = 1\text{M}$ 电影 (每部 1GB)，$n = 1\text{M}$ 节点 (每个节点的容量为 1TB)，则每个节点可以存储 $1\text{TB}/1\text{GB} = 1\text{K}$ 部电影。这样，我们可以采用 $k = 1\text{K}$ 哈希函数。根据定理 8.2，每个节点平均存储大约 1K 部电影。
- 采用下面的切诺夫界 (Chernoff Bound)，并且有 $\mu = km/n = 1\text{K}$，我们可以得出结论：一个节点所使用的存储空间超出平均期望 10% 的概率小于 1%。[2]

**事实** 8.3. 根据某个版本的 **Chernoff Bound**，我们有:
若 $x_1, \ldots, x_n$ 是独立同分布的随机变量，且均服从下面的伯努利分布 (Independent Bernoulli-Distributed Random Variables): $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, 则，对 $X := \sum_{i=1}^{n} x_i$ 以及 $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$，有如下关系:

$$\text{对任意的}\delta > 0\text{: } Pr[X \geqslant (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

[1]译者注：即 $E(X+Y) = E(X) + E(Y)$。

[2]译者注：注意到 $x_i$ 为将第 $i$ 部电影存放于该节点的随机事件，则 $X := \sum_{i=1}^{n} x_i$ 为该节点上存储的电影总数。将 $\mu = 1000$, $\delta = 0.1$ 代入式子中即可得到结论。

**评论:**

- 我们可以直观地将电影直接存放在算法 8.1 所建议的节点上，当然我们也可以将电影存放在任何我们希望的地方，而只将电影的指针 (即电影的实际存储位置) 存放在算法 8.1 所建议的节点上。

- 本章中，我们将讨论极端的不可靠情况。假设节点们平均只能可靠地工作一个小时，我们该怎么做？换句话说，节点们在很频繁地*搅动* (Churn)，它们不停地加入或离开这个分布式系统。

- 在如此频繁的搅动之下，每秒钟都有成百上千个节点变动。没有一个节点可以精确地掌握当前系统中还存在哪些节点。这个场景和经典的分布式系统存在显著的差别。在经典的分布式系统中，单节点故障都已经是一个小型的灾难了，因为所有其他节点都不得不重新建立一个一致的系统视图 (定义 5.4)。在高度搅动的情景下，在任何时刻都不可能拥有一个一致的视图。

- 相反地，每个节点将只了解一个很小的“邻居”集合 (通常小于 100 个节点)。以这样的方式，节点们可以抵挡高度搅动带来的影响。

- 这样做 (仅了解邻居的情况) 的负面影响是，节点们将不能直接知道哪个节点存储了哪部电影。相反地，节点在搜索一部电影时，将不得不询问邻居，而邻居再递归地询问它的邻居，直到找到存放这部电影的节点为止。于是在这样的分布式系统中的所有节点就构成了一个虚拟的网络，也被称为一个*覆盖网络* (Overlay Network)。

## 8.2 超立方体网络 (Hypercubic Networks)

本节中，作为扩展知识，我们将介绍一些覆盖拓扑结构 (Overlay Topologies)。

**定义** 8.4 (**拓扑属性**). 虚拟网络应该具备下面的性质：

- 网络在某种程度上是同质的 (Homogeneous)，即：不存在一个节点在网络中处于支配地位，也不存在一个节点可以造成单点故障。
- 节点都有 *ID*，而且所有的 ID 值的范围在 $[0,1)$。这样我们才可以采用算法 8.1，利用哈希编码来存储数据。
- 每个节点的度数都比较小，均为网络节点数 $n$ 的多重对数 (Polylogarithmic) 量级。这可以使每个节点和它的邻居维持持久的连接，从而应付搅动 (Churn) 现象。
- 网络应当具有较小的直径 (Diameter)[3]，且易于实现在网络中的路由。如果一个节点不清楚某个数据项的信息，它应该知道去询问哪个邻居。通过少量的跳转 ($n$ 的多重对数级别)，一个节点应当能找到存储正确信息的目标节点。

**评论:**

- 常用的网络拓扑包括树 (Trees)，环 (Rings)，网格 (Grids)，以及圆环 (Tori)。这些基本的拓扑可以组合或衍生出更多的网络。
- 树的优点在于很容易路由。从任意源节点到任意目标节点都只有一条路径。然而，每棵树的根节点都是一个瓶颈，因此树并非同质的 (Homogeneous)。如果希望去掉瓶颈，可采用肥胖树

[3]译者注：网络直径是指网络中任意两节点间距离的最大值。

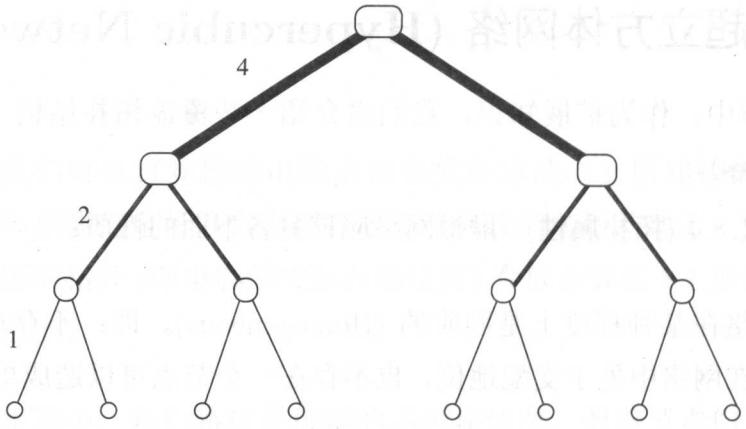



图 8.5: 一棵肥胖树 (Fat Tree) 的结构

(Fat Tree)。肥胖树的特点在于：假设 $e$ 是一条边，连接了一个节点 $v$ 到它的父节点 $u$。则 $e$ 的容量 (Capacity) 与以 $v$ 为顶点的子树的叶节点数量成正比。图 8.5给出了肥胖树的一个例子。图中的边上的数字表示该边的容量。

- 实际中，更倾向于构造具有均匀容量的网络，比如网格或圆环。下文中，除非特别说明，我们将默认所有的边的容量均为 1.

**定义** 8.6 (**网眼** (Mesh), **圆环** (Torus),). 设 $m, d \in \mathbb{N}$。一个$(m,d)$-*mesh* $M(m,d)$ 是一个图，包括节点集合 $V = [m]^d$，以及边的集合

$$E = \left\{ \{(a_1, \ldots, a_d), (b_1, \ldots, b_d)\} \mid a_i, b_i \in [m], \sum_{i=1}^{d} |a_i - b_i| = 1 \right\}$$

这里 $[m]$ 表示集合 $\{0, \ldots, m-1\}$。

一个$(m,d)$-*torus* $T(m,d)$ 是一个图，包含一个 $(m,d)$-mesh，以及附加的环绕边，这些边连接下面两组节点 $(1)(a_1, \ldots, a_{i-1}, m-1, a_{i+1}, \ldots, a_d)$;$(2)(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_d)$。

这里，$i \in \{1, \ldots, d\}$，且 $a_j \in [m]$，满足条件 $j \neq i$。也就是说，前面在求和算子中的表达式 $a_i - b_i$ 用来计算两个相邻顶点坐标差值。

$M(m,1)$ 也被称为一条路径 (path)，$T(m,1)$ 是一个环 (cycle)，而 $M(2,d) = T(2,d)$ 是一个 $d$-维超立方体 ($d$-Dimensional Hypercube)。图 8.7 展示了一个线性数组，一个圆环，以及一个超立方体。

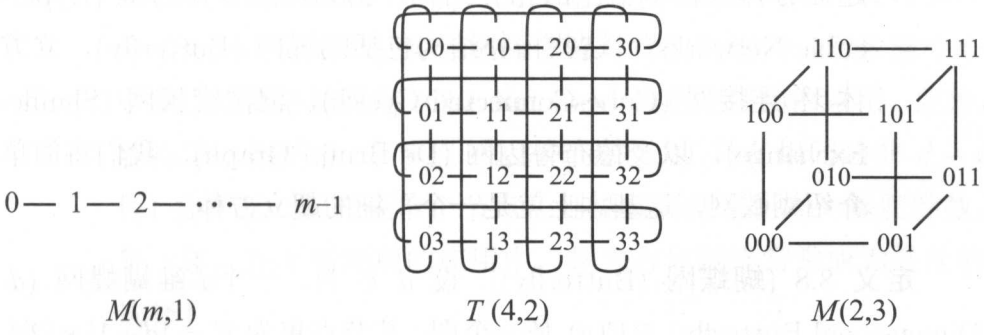



图 8.7: $M(m,1)$, $T(4,2)$, 以及 $M(2,3)$ 的结构

**评论:**

- 在一个网眼，圆环，或超立方体上路由是非常自然的。在一个 $d$-维超立方体上，从一个源节点 $s$ 到一个目标节点 $t$ [4]，我们只需要更正所有不同的数字，每次更正一个。也就是说，如果源点和目标点的差异为 $k$ 个数字，将有 $k!$ 条不同的路径，每条路径有 $k$ 跳。
- 根据定义 8.4，节点的 $d$-位 ID 需要被映射到 $[0,1)$ 空间。一个简单的办法是将 ID 视为一个十进制数的小数部分的二进制表示。比如，一个 ID **101** 被转换为 $\mathbf{0.101_2}$，其十进制的数值为 $\mathbf{0} \cdot 2^0 + \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} = \frac{5}{8}$。

[4]译者注：这里，$s$ 和 $t$ 是节点的坐标，即一组数字构成的字符串。

- 弦 (Chord) 结构是超立方体的近亲，基本可以将弦视为一个不那么严格的超立方体。超立方体中，任意下面的节点对之间存在连接：ID 在 $[0,1)$ 范围内，彼此 ID 之间的距离正好等于 $2^{-i}$, $i = 1, 2, \ldots, d$ in $[0,1)$。而弦中将连接距离近似为 $2^{-i}$ 的节点对。
- 超立方体有很多衍生的拓扑结构，称为*超立方体网络* (Hypercubic Networks)。这些拓扑结构包括蝴蝶网 (Butterfly)，立方体-环-连接网 (Cube-Connected-Cycles)，混洗交换网 (Shuffle-Exchange)，以及德布鲁因网 (De Bruijn Graph)。我们将简单介绍蝴蝶网，这基本上就是一个平铺的超立方体。

**定义** 8.8 (**蝴蝶网** (Butterfly)). 设 $d \in \mathbb{N}$。一个$d$-维蝴蝶网 ($d$-Dimensional Butterfly) $BF(d)$ 是一个图，其节点集为 $V = [d+1] \times [2]^d$，边集为 $E = E_1 \cup E_2$，其中：

$$E_1 = \{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2]^d\},$$

$$E_2 = \{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2]^d, |\alpha - \beta| = 2^i\}$$

我们称一个节点集合 $\{(i,\alpha) \mid \alpha \in [2]^d\}$ 处于蝴蝶网的 第 $i$ 层。而一个 $d$-维环绕蝴蝶网 ($d$-Dimensional Wrap-Around Butterfly) W-BF($d$)，是在一个 $BF(d)$ 上增加一些边：$(d,\alpha) = (0,\alpha)$ $(\forall \alpha \in [2]^d)$。

**评论:**

- 图 8.9 展示了一个 3-维蝴蝶网 $BF(3)$。一个 $BF(d)$ 拥有 $(d+1)2^d$ 个节点, $2d \cdot 2^d$ 条边，且度数为 4[5]。若将节点集 $\{(i,\alpha) \mid i \in [d]\}$ $(\forall \alpha \in [2]^d)$ 组成一个单一的节点，则得到了一个超立方体。

[5]译者注：最大度数，如图 8.9 中第 2 层的节点。

- 蝴蝶网的优点在于节点的度数为恒定的，而超立方体可以具备更好的容错性。
- 其实类似蝴蝶网的拓扑结构在实践中很常见，比如排序网络 (Sorting Network)，通信交换 (Communication Switches)，数据中心网络 (Data Center Networks)，以及快速傅立叶变换 (Fast Fourier Transform, FFT)。无线通信中的重要网络，贝内斯网络 (Benes Network) 实际是两个背对背的蝴蝶网。而数据中心中常用的克洛斯网络 (Clos Network) 也是蝴蝶网的近亲。此外，若将第 $i$ 层的 $2^i$ 个节点 (这些节点的 ID 具有相同的前 $d-i$ 位) 合并在一起构成单个节点，则蝴蝶网就退化为一棵肥胖数。每一年，在不断涌现的应用中，超立方体网络常常成为最佳的选择！
- 下面我们将定义立方体-环-连接网 (Cube-Connected-Cycles Network)。这种网络的度数为 3，它实际是将超立方体的拐角替换为圆环得到的。

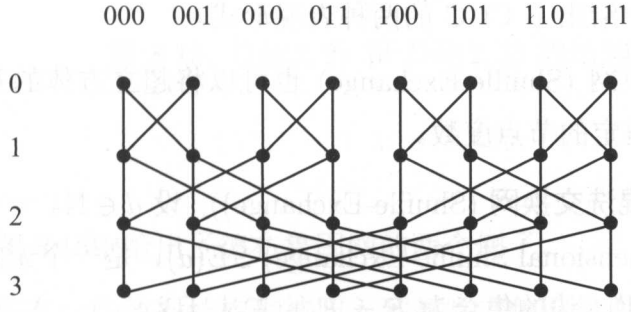



图 8.9: BF(3) 的结构图

**定义** 8.10 (**立方体-环-连接网** (Cube-Connected-Cycles)). 设 $d \in \mathbb{N}$。一个立方体-环-连接网 (Cube-Connected-Cycles Network) CCC($d$)，是

一个图，其节点集为 $V = \{(a,p) \mid a \in [2]^d, p \in [d]\}$，边的集合为

$$\begin{aligned} E &= \big\{\{(a,p),(a,(p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\big\} \\ &\quad \cup \big\{\{(a,p),(b,p)\} \mid a,b \in [2]^d, p \in [d], a = b,\ a_p \text{ 除外}\big\} \ . \end{aligned}$$

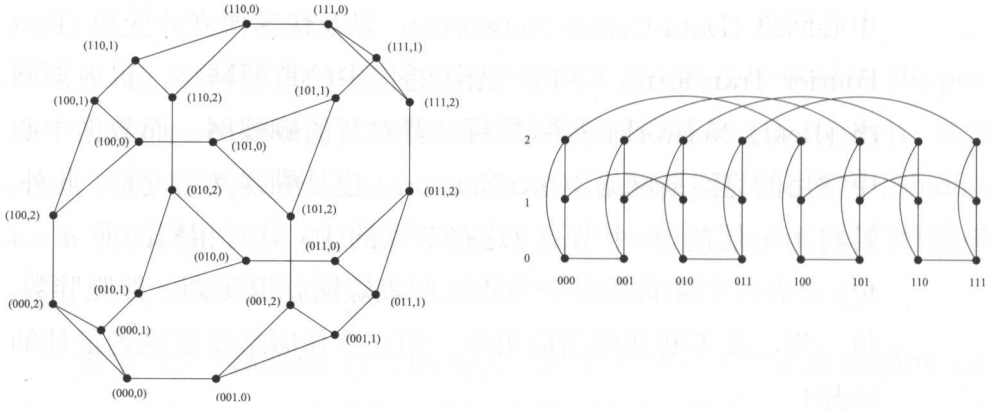



图 8.11: CCC(3) 的结构.

**评论:**

- 图 8.11中给出了 CCC 的两种表现形式。
- 混洗交换网 (Shuffle-Exchange) 也可以将超立方体的互联结构转换为恒定的节点度数。

**定义** 8.12 (**混洗交换网** (Shuffle-Exchange)). 设 $d \in \mathbb{N}$。一个$d$-维混洗交换网 ($d$-Dimensional Shuffle-Exchange) $SE(d)$，是一个无向图，其节点集为 $V = [2]^d$，边的集合为 $E = E_1 \cup E_2$，且：

$$E_1 = \{\{(a_1,\ldots,a_d),(a_1,\ldots,\bar{a}_d)\} \mid (a_1,\ldots,a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\},$$

$$E_2 = \{\{(a_1,\ldots,a_d),(a_d,a_1,\ldots,a_{d-1})\} \mid (a_1,\ldots,a_d) \in [2]^d\} \ .$$

图 8.13 展示了一个 3-维和一个 4-维的混洗交换网。

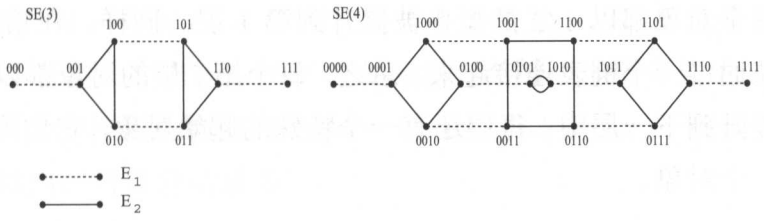



图 8.13: SE(3) 和 SE(4) 的结构

**定义** 8.14 (**德布鲁因网** (DeBruijn)). 一个 $b-$ 叉 $d-$ 维的德布鲁因网 ($b$-ary DeBruijn Graph of Dimension $d$) $DB(b,d)$ 是一个无向图 $G=(V,E)$，其节点集合为 $V=\{v\in[b]^d\}$，边的集合为 $E$，若一条边 $\{v,w\}$ 具备下面的性质，则该边被包含在 $E$ 中：$w\in\{(x,v_1,\ldots,v_{d-1}):x\in[b]\}$, 此处 $v=(v_1,\ldots,v_d)$.

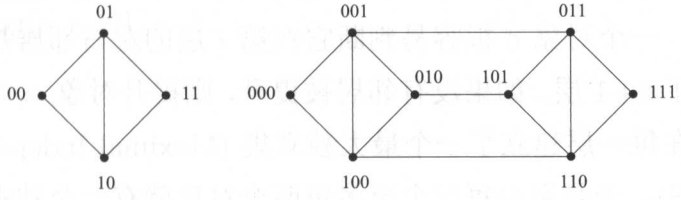



图 8.15: $DB(2,2)$ 和 $DB(2,3)$ 的结构

**评论:**

- 图 8.15中给出了德布鲁因网的两个例子。
- 还有一些数据结构也被归入到超立方体网络中，比如跳表。跳表实际是为懒惰的程序员准备的平衡二叉搜索树 (Balanced Binary Search Tree)。

**定义** 8.16 (**跳表** (Skip List)). 跳表 (Skip List) 是一个普通的有序链表，加上一些额外的前向链接。普通的链表是跳表的第 0 层。此外，

链表中每个对象都以 1/2 的概率被提升到第 1 层。同样，在第 1 层中，所有对象通过一个链表连接起来。总之，每个第 $i$ 层的对象都以 1/2 的概率被提升到下一层中。每层还有一个特殊的起始对象，它们均指向每层的第一个对象。

**评论:**

- 跳表中，查找、插入和删除操作都可以在 $\mathcal{O}(\log n)$ 的平均时间内完成：只需要在错过搜索的位置时从高层跳到低层。每个对象所耗费的存储成本也是常数，因为平均来说一个对象只有两个前向链接。
- 可以很容易地去掉随机性，在第 $i$ 层，按某个固定的策略选取一定比例的对象，并把它们提升到第 $i+1$ 层。当插入或删除时，一个对象 $o$ 很容易判断它在第 $i$ 层的左右邻居是否被提升到了 $i+1$ 层。如果没有邻居被提升，则提升对象 $o$。本质上，我们在每一层建立了一个最大独立集 (Maximal Independent Set, MIS)。于是至少每三个至多每两个对象就有一个被提升。
- 有很多跳表的变种，比如跳表图 (Skip List Graph)。在这个结构中，我们不再提升一半的节点到下一层，相反地，我们总是提升所有的节点。这个做法类似于平衡二叉树。所有的节点都归属于根节点。在每一层，一半节点被提升到左边，另一半节点被提升到右边。于是在第 $i$ 层，我们拥有 $2^i$ 个表 (或环)，包含了大约 $n/2^i$ 个对象。跳表图具备定义 8.4描述的所有性质。
- 更一般地，定义 8.4 中度数和直径之间有什么关系呢？下面的定理给出了一个通用的下界。

**定理** 8.17. 每个图，若节点数为 $n$，且节点的最大度数为 $d > 2$，则其直径至少为 $\lceil(\log n)/(\log(d-1))\rceil - 2$。

证明. 假定有一个图 $G = (V, E)$ 其节点个数为 $n$，且节点最大度数为 $d$。从图中任意一个节点 $v \in V$ 开始漫游，在第一步最多到达 $d$ 个节点，在第二步最多再到达额外的 $d \cdot (d-1)$ 个节点。那么，一般而言，在最多 $r$ 步以内，可以到达最多

$$1 + \sum_{i=0}^{r-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^r - 1}{(d-1) - 1} \leqslant \frac{d \cdot (d-1)^r}{d-2}$$

个节点 (包括 $v$)。若 $r$ 为图的直径，则上面的式子应至少为 $n$，这样 $v$ 才可以在 $r$ 步以内到达图中任意一个节点。于是我们有

$$(d-1)^r \geqslant \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad r \geqslant \log_{d-1}((d-2) \cdot n/d) .$$

当 $d > 2$ 时，我们可以[6]得到 $\log_{d-1}((d-2)/d) > -2$，因此 $r \geqslant \lceil (\log n)/(\log(d-1)) \rceil - 2$。

**评论:**

- 换句话说，若超立方体网络的度数为常数，则其直径可以达到渐近最优的 (Asymptotically Optimal) 的值 $D$。
- 其他超立方体图 (网络) 都是在度数 $d$ 和最优的直径 $D$ 之间权衡。比如，煎饼图 (Pancake Graph)，满足 $\max(d, D) = \Theta(\log n / \log\log n)$。一个 $d$-维的煎饼图中，一个节点 $u$ 的 ID 是在集合 $1, 2, \ldots, d$ 上的任意一个排列。图中任意两个节点 $u, v$，若将 $u$ 的 ID 的前 $i$ 位 $(2 \leqslant i \leqslant n)$ 反转之后得到 $v$ 的 ID，则 $u, v$ 之间存在一条边。比如，若 $d = 4$，节点 $u = 2314$ 和 $v = 1324$ 相邻。

[6]译者注：只需要证明 $(d-2)/d > 1/(d-1)^2 \Leftrightarrow (d-2)(d-1)^2 > d \Leftrightarrow (d-1)^2 > d$，最后一个不等式可以根据一元二次方程得证。

- 还有一些图，它们虽然不是超立方体网络，但却具备定义 8.4中的部分性质。比如，社交网络中的重要模型，小世界网络 (Small-World Graph)，也有很小的网络直径。但是，和超立方体网络不同的是，小世界网络不是同质的，部分节点具有很大的度数。
- 扩展图 (Expander Graphs) 是一种稀疏图，且具备很好的连通性，从每一个不大的节点集合出发，可以连接到很大的节点集合。扩展图是同质的，度数和直径都较小。但是扩展图经常是不能路由的。

## 8.3 DHT & Churn

**定义** 8.18 (**分布式哈希表** (Distributed Hash Table, DHT) ). 一个分布式哈希表 (Distributed Hash Table, DHT) 是一个实现了分布式存储的分布式数据结构。一个 DHT 应该至少支持 (i) (根据键的) 搜索操作；(ii) 插入 (键，值) 对操作；(iii) 删除 (键) 操作。

**评论:**

- 除了存储电影之外，DHT 还有很多应用，比如域名系统 (DNS) 本质上就是一个 DHT。
- DHT 可以采用超立方体覆盖网络 (Hypercubic Overlay Network) 来实现，只有其节点的 ID 取值在 $[0,1)$ 的范围。
- 一个超立方体 (Hypercube) 可以直接用作 DHT。只需要使用一个全局的哈希函数集合 $h_i$，将电影名称映射为长度为 $d$ 的字符串。
- 使用其他超立方体结构 (Hypercubic Structures) 来作为 DHT 相对较为复杂。比如，使用蝴蝶网时，可能需要直接使用 $d+1$

层来作为副本，也就是说，所有在 $d+1$ 层上的节点都负责同一个 ID。

- 其他的一些超立方体网络，比如煎饼图，可能需要一些扭转 (Twisting) 来找到合适的 ID。
- 我们假定一个节点在加入网络时知道另一个已经在网络中的节点。这实际是一个引导问题 (Bootstrap Problem)。典型的解决方案是，如果一个节点曾经和 DHT 有过连接，那么就直接尝试之前已知的部分节点。否则，新加入的节点需要向某个权威节点请求一个 IP 地址列表，里面列出了已经在 DHT 中的节点。
- 有很多工作分析 DHT 在面临攻击者时的场景，这些攻击者可以从网络中随机选择一部分节点，使它们崩溃。而在一些节点崩溃之后，系统被允许在较为充分的时间内恢复。但是，这个设定明显并不符合实际情况。而本节所讨论的场景与此显著不同，其区别主要在于以下两个方面。

  - 首先，我们假定最坏的一种“节点加入和离开网络”情况。我们考虑一个攻击者可以删除或增加有限数量的节点，该攻击者可以主动选择[7]一部分节点崩溃，以及节点加入网络的方式。
  - 此外，攻击者不需要等待系统恢复之后才破坏另一部分节点。与之相反，攻击者可以持续地使节点崩溃，而系统始终在尝试存活。事实上，系统从未全部恢复，但却始终可用。特别地，系统可以防范这样的攻击者：它持续地攻击系统的“薄弱部位”。比如，攻击者可以在 DHT 中插入一

[7]译者注：与上面的场景不同，不是网络中随机选择的节点。攻击者可以选择部分处于关键位置的节点。

个爬虫，学习网络的拓扑结构，然后重复破坏特定的一组节点，以达到使 DHT 分区 (Partition)[8]的企图。系统的应对措施是持续将一些未被攻击的节点移动到这些受到攻击的区域，或是将新加入的节点置于这个区域。

- 显然，我们不能允许攻击者具备无限的能力。例如，在一个固定的时间段内，攻击者最多可以增加/删除 $O(\log n)$ 个节点，这里 $n$ 是当前系统中节点总数。这个模型涵盖了一类常见的攻击场景：攻击者们通过分布式的拒绝服务攻击 (Distributed Denial of Service Attack) 持续地破坏一部分节点，但是每次能攻击的节点数为 $O(\log n)$。为实现这个模型，相应的算法要求消息及时地传输，每两个节点之间消息传输时间不能超过一个预设的常数，也就是说，同步模型。可以采用一个普通的同步装置来实现这个算法。我们只需要消息传输时间在预设的范围内，从而可以根据传世来判断攻击行为。而每轮的持续时间则与最慢消息的传输耗时成正比。

**评论:**

- 我们看看每个节点的邻居数：每个节点拥有对数级的超节点邻居，每个超节点邻居又包含对数级的节点，于是每个节点的邻居数为 $\Theta(\log^2 n)$。然而，采用一些额外的优化手段，我们可以使得每个节点的邻居数降到 $\Theta(\log n)$。
- 超节点大小 (即每个超节点所包含的节点数) 的均衡性实际是在超立方体上的动态令牌分布问题 (a Dynamic Token Distribution Problem on the Hypercube)。每个超节点拥有特定数量

[8]译者注：即不再连通。

**算法 8.19** DHT

1: 给定：一组全局的哈希函数 $h_i$，以及一个超立方体 (或者任何超立方体网络)。
2: 每个超立方体虚节点 (“超节点”) 包含 $\Theta(\log n)$ 个节点。
3: 每个节点均与其所在的超节点以及相邻的超节点内所有节点相连。
4: 由于搅动 (Churn)，部分节点不得不转到其他超节点。因此，最终所有的超节点在任何时候都拥有相同的一组节点，而这组节点占所有节点的比例为一个常数。
5: 如果网络中节点的总数 $n$ 增加或减少的幅度超过一个特定的阈值，超立方体将相应地增加或减少 1 个维度。

的令牌，目标是将这些令牌沿着图上的边进行分配，使得所有的超节点拥有大致相同的令牌数。当令牌在图中移动时，一个攻击者可以持续地插入或删除令牌。如图 8.20所示。

- 总之，上述存储系统基于两个基本的组件：(i) 一个算法，实现了上述动态令牌分发过程；(ii) 一个信息聚合算法，用来估计系统中节点的数量并且相应地调整超立方体的维度。

**定理** 8.21 (**带搅动的** DHT). 上述系统是一个完全可扩展的，高效率的分布式存储系统，且可以容忍最坏情况下单位时间内 $O(\log n)$ 次节点加入或删除。而在其他的存储系统中，若节点拥有 $O(\log n)$ 个覆盖邻居，常用的操作 (如查找、插入) 须耗时 $O(\log n)$。

**评论:**

- 事实上，处理搅动仅仅是对分布式存储系统最低的要求。前沿的工作已提出了更精妙的体现结构，这些结构可以处理其他安全问题，比如隐私或者是拜占庭攻击。

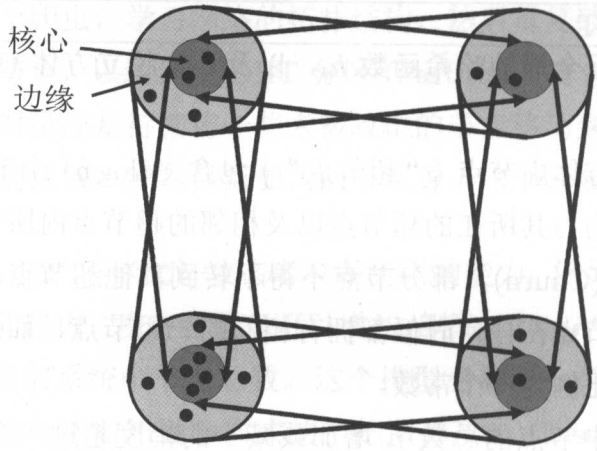



图 8.20: 图中模拟了一个 2-维的超立方体，有 4 个超节点，每个超节点由多个节点构成。所有的节点位于超节点的核心或者是边缘。在同一个超节点的所有节点彼此相连，此外，所有节点都连接到邻居超节点的核心节点上。只有核心节点存储数据，而边缘节点在超节点之间移动以缓解攻击者所造成的搅动。

## 章节说明

分布式存储的思想出现在 2000 年左右，正处于 P2P 文件共享的热潮中，因此这个领域很多重要的研究工作都打上了 P2P 的标签。Plaxton, Rajaraman, 以及 Richa 的论文 [PRR97] 展示了结构化 P2P 体系架构的蓝图，这个框架下的工作包括 Chord [SMK+01], CAN [RFH+01], Pastry [RD01], Viceroy [MNR02], Kademlia [MM02], Koorde [KK03], SkipGraph [AS03], SkipNet [HJS+03], 以及 Tapestry [ZHS+04].

但 Plaxton 等人的文章也是站在前面巨人的肩膀上的，比如线性且一致性哈希 (Linear and Consistent Hashing) [KLL+97], 定位共享对象 (Locating Shared Objects) [AP90, AP91], 简洁路由 (Compact Routing)

[SK85, PU88], 以及更早的工作，比如超立方体网络 [AJ75, Wit81, GS81, BA84].

此外，在基于前缀的覆盖结构中使用的技术和另一个名为 LAND 的工作相关，这是一个位置感知的分布式哈希表，由 Abraham 等人提出 [AMD04].

近来，很多 P2P 研究更侧重于安全方面，比如攻击 [LMSW06, SENB07, Lar07], 以及可证明的对抗措施 (Provable Countermeasures) [KSW05, AS09, BSS09]。另一个当前热门的话题是使用 P2P 来帮助大规模分发直播视频流 [LMSW07]。在 P2P 计算方面，有不少值得推荐的书籍，比如 [SW05, SG05, MS07, KW08, BYL08]。

# 参考文献

[AJ75] George A. Anderson and E. Douglas Jensen. Computer Interconnection Structures: Taxonomy, Characteristics, and Examples. *ACM Comput. Surv.*, 7(4):197–213, December 1975.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: stretch (1 + epsilon) locality-aware networks for DHTs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 550–559, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[AP90] Baruch Awerbuch and David Peleg. Efficient Distributed Construction of Sparse Covers. Technical report, The Weizmann Institute of Science, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent Online Tracking of Mobile Users. In *SIGCOMM*, pages 221–233, 1991.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *SODA*, pages 384–393. ACM/SIAM, 2003.

[AS09] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. *Theory Comput. Syst.*, 45(2):234–260, 2009.

[BA84] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. Comput.*, 33(4):323–333, April 1984.

[BSS09] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A DoS-resilient information system for dynamic data management. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 300–309, New York, NY, USA, 2009. ACM.

[BYL08] John Buford, Heather Yu, and Eng Keong Lua. *P2P Networking and Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[GS81] J.R. Goodman and C.H. Sequin. Hypertree: A Multiprocessor Interconnection Topology. *Computers, IEEE Transactions on*, C-30(12):923–933, dec. 1981.

[HJS+03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: a scalable overlay

network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, 2003.

[KLL+97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Frank Thomson Leighton and Peter W. Shor, editors, *STOC*, pages 654–663. ACM, 1997.

[KSW05] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*, February 2005.

[KW08] Javed I. Khan and Adam Wierzbicki. Introduction: Guest editors' introduction: Foundation of peer-to-peer computing. *Comput. Commun.*, 31(2):187–189, February 2008.

[Lar07] Erik Larkin. Storm Worm's virulence may change tactics. http://www.networkworld.com/news/2007/080207-

black-hat-storm-worms-virulence.html, Agust 2007. Last accessed on June 11, 2012.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LMSW07] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus*, September 2007.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[MS07] Peter Mahlmann and Christian Schindelhauer. *Peer-to-Peer Networks*. Springer, 2007.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *SPAA*, pages 311–320, 1997.

[PU88] David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 43–52, New York, NY, USA, 1988. ACM.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[SENB07] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting KAD: possible uses and misuses. *SIGCOMM Comput. Commun. Rev.*, 37(5):65–70, October 2007.

[SG05] Ramesh Subramanian and Brian D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology.* IGI Publishing, Hershey, PA, USA, 2005.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and Implicit Routing in Networks. *Comput. J.*, 28(1):5–8, 1985.

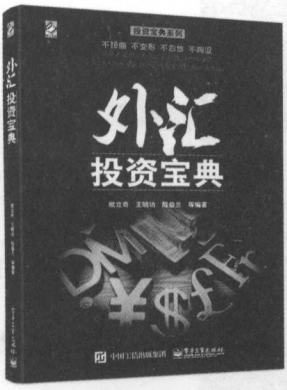
[SMK+01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

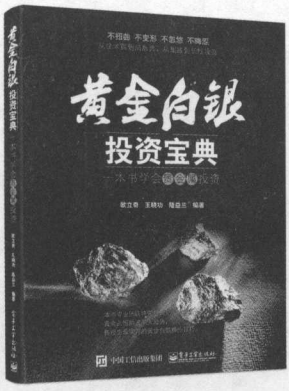
[SW05] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

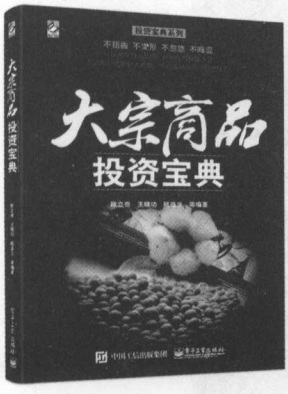
[Wit81] L. D. Wittie. Communication Structures for Large Networks of Microcomputers. *IEEE Trans. Comput.*, 30(4):264–273, April 1981.

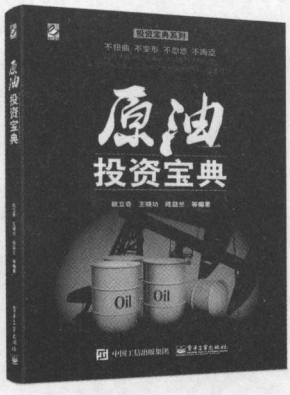
[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

# 电子工业出版社好书分享




整套丛书洞见贵金属投资、原油交易、外汇投资、大宗商品交易大趋势，从技术面到消息面，从短线投资到长线投资，以完整严密的知识体系、幽默诙谐的语言，传授实用的操作技巧！

每本书都包含近百经典真实案例，让读者一眼透过现象看到本质，举一反三，活学活用！

## 关于作者

Roger Wattenhofer博士是瑞士苏黎世联邦理工学院（ETH Zurich）的一名教授。在这之前，他曾在美国布朗大学（Brown University）及微软研究院工作。他的研究兴趣主要包括容错分布式系统、高效的网络算法，以及加密货币。截至本书出版，他已发表了250多篇学术论文。

## 关于本书

金融科技（FinTech）的研发人员和经理们认为区块链具备颠覆整个金融世界的潜力。区块链使得一个分布式系统的参与者们以可靠的方式就系统的全局状态达成一致，并追踪在系统中发生的改变。虽然“区块链”这个词来源于比特币网络，但在比特币或其他加密货币问世之前，共识技术就早已备受分布式系统领域的学者关注了。存在各种各样解决共识问题的概念和协议，各有其优缺点。

本书用一种科学的方式来介绍构建一个容错的分布式系统的基础技术。书中介绍了不同的协议和算法，它们是容错操作的基础。此外本书还讨论实现了这些技术的实际系统。

欢迎投稿联络：高洪霞
邮箱：gaohx@phei.com.cn

# 区块链
# 核心算法解析

## 本书所获赞誉

介绍区块链应用的书籍非常多，而从理论、技术层面介绍区块链的书比较少。很高兴看到有这样一本从理论、技术层面介绍区块链的书籍出版。希望大家能耐心读读这本书，更深入地理解区块链技术，从而有助于推动区块链技术的发展和应用。

——高卢麟博士 中国互联网协会副理事长，美国芝加哥马歇尔法学院客座教授

本书着眼于区块链的核心问题——拜占庭共识，针对不同的应用场景，介绍了适用的分布式共识算法。书中包含了很多算法及证明，深入剖析了共识算法的核心思想。译者除原稿翻译之外，还增加了译者自己不少的注释，对书中的算法、公式进行注解。另外，书中还单独增加了两章新的内容。一章是介绍Paxos算法的发展史和在工业界的应用情况，另一章是对比分析当前主流的两个共识机制，比特币的PoW和私有链的PBFT。现在都讲究“混搭”，这本译著也是一种形式的混搭。

——杜小勇 中国计算机学会数据库专委会主任，教育部数据工程与知识工程重点实验室主任

《区块链核心算法解析》以共识机制为主体，系统介绍了区块链所涉及的各种关键定理和证明，也给出了相应算法。难能可贵的是，作者还结合实例讲述了不同场景下的共识机制的设计方法。这是一本关于区块链核心技术的系统论著，对于区块链科研和应用人员都具有很高的参考价值。

——戴斌 国防科技大学机电工程与自动化学院副总工程师

上架建议：区块链

ISBN 978-7-121-31328-8

9 787121 313288 >

定价：59.00元

责任编辑：高洪霞
封面设计：李　玲

PHEI